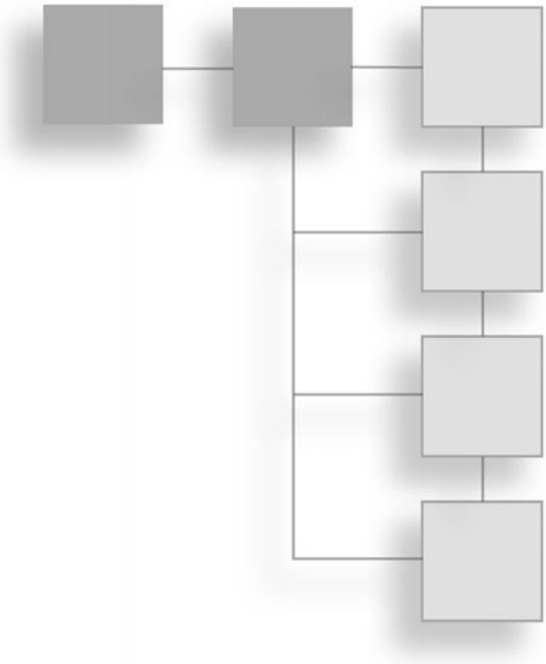
VISIT…

COURSE TECHNOLOGY
CENGAGE Learning
Professional • Technical • Reference

2D GAME BUILDING

FOR teens

MICHAEL DUGGAN



2D Game Building for Teens

Michael Duggan

Course Technology PTR
A part of Cengage Learning

COURSE TECHNOLOGY
CENGAGE Learning™

Australia • Brazil • Japan • Korea • Mexico • Singapore • Spain • United Kingdom • United States

COURSE TECHNOLOGY
CENGAGE Learning

2D Game Building for Teens
Michael Duggan

Publisher and General Manager, Course Technology PTR: Stacy L. Hiquet

Associate Director of Marketing: Sarah Panella

Manager of Editorial Services: Heather Talbot

Marketing Manager: Jordan Casey

Senior Acquisitions Editor: Emi Smith

Project Editor: Jenny Davidson

Technical Editor: Ben Garney

PTR Editorial Services Coordinator: Jen Blaney

Interior Layout Tech: Macmillan Publishing Solutions

Cover Designer: Mike Tanamachi

CD-ROM Producer: Brandon Penticuff

Indexer: Sharon Shock

Proofreader: Sara Gullion

© 2009 Course Technology, a part of Cengage Learning.

ALL RIGHTS RESERVED. No part of this work covered by the copyright herein may be reproduced, transmitted, stored, or used in any form or by any means graphic, electronic, or mechanical, including but not limited to photocopying, recording, scanning, digitizing, taping, Web distribution, information networks, or information storage and retrieval systems, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without the prior written permission of the publisher.

For product information and technology assistance, contact us at **Cengage Learning Customer & Sales Support, 1-800-354-9706**

For permission to use material from this text or product, submit all requests online at **cengage.com/permissions**
Further permissions questions can be e-mailed to **permissionrequest@cengage.com**

Torque Game Engine is the trademark of GarageGames.com, Inc. GarageGames is a registered trademark of GarageGames.com, Inc. Torque Game Builder version 1.74, Torsion syntax editor, the Platformer Starter Kit, Jewel Quest II, King Kong Skull Island Adventure, Magic Pearls, Phantasia II, Pirate Tales, Puzzle Poker, Rack Em Up Road Trip, and Tank Buster are registered trademarks or trademarks of GarageGames.

All other trademarks are the property of their respective owners.

Library of Congress Control Number: 2008929245

ISBN-13: 978-1-59863-568-3

ISBN-10: 1-59863-568-9

eISBN-10: 1-59863-741-x

Course Technology, a part of Cengage Learning
20 Channel Center Street
Boston, MA 02210
USA

Cengage Learning is a leading provider of customized learning solutions with office locations around the globe, including Singapore, the United Kingdom, Australia, Mexico, Brazil, and Japan. Locate your local office at: **international.cengage.com/region**

Cengage Learning products are represented in Canada by Nelson Education, Ltd.

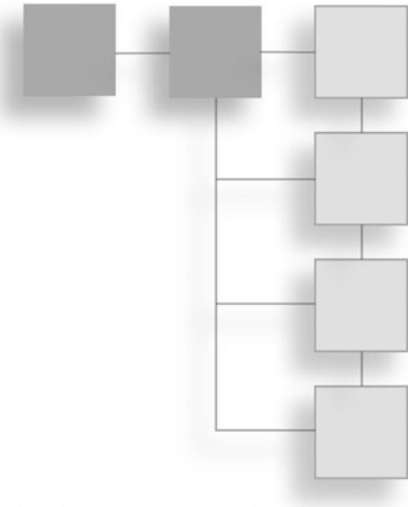
For your lifelong learning solutions, visit **courseptr.com**

Visit our corporate website at **cengage.com**

Printed in Canada
1 2 3 4 5 6 7 11 10 09

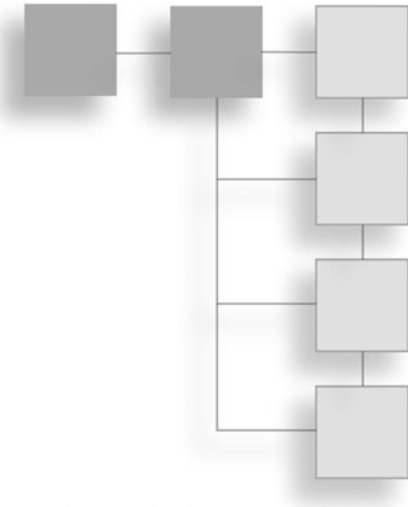
To the people who reflect the seasons of my life and make me see the world with new eyes.

Acknowledgments

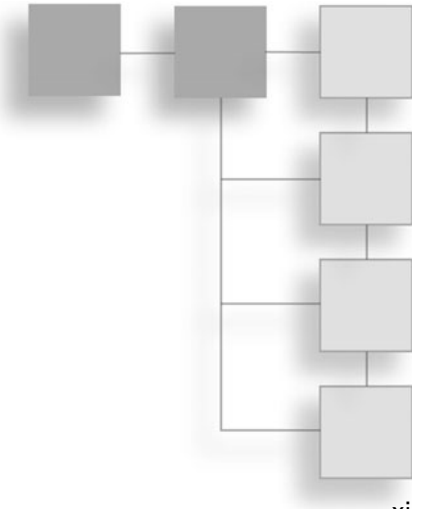


Thanks to Wade Tinney and Kevin Ryan for the help they gave me. Thanks to my editors Emi Smith, Jenny Davidson, and Ben Garney for helping me put this book together and making sure I didn't look like a fool. Finally, thanks to the kind folks at GarageGames for their continued support. As the saying goes, it takes a village, not one person, to write a book.

About the Author



Mike Duggan is a college teacher in visual arts, graphic/web design, and game design. He designed core curriculum for the gaming and robotics program at Bryan College, based in Kansas City, Missouri, and he has been a guest speaker on the topic of game engine technology at several conferences. He is the author of *The Official Guide to 3D GameStudio*, *Torque for Teens*, and *Web Comics for Teens*. Mike spends most of his free time drawing cartoons and making animations for games. For more information about the author, go to http://www.mdduggan.com.



Contents

Introduction xi

Chapter 1 Want to Make Games? 1

What Are Games? 2
- Huizinga's Magic Circle 3
- Core Mechanics 3
- Gameplay 4
- Graphics 6

How to Be a Game Designer 7
Come Up with Your Own Game Ideas 12
- Write Your Ideas Down 15

Game Design Step-by-Step 19
- Preproduction 19
- Production 23
- Postproduction 24

Starting Your Own Game Company 26
The Importance of Being Independent 26
Review 28

Chapter 2 Brief Look at 2D Games 29

Where Do 2D Games Come From? 30
- Electronic Game Firsts 30
- 2D in a 3D World 35

Popular 2D Game Genres 38
Action Games 38
Adventure Games 39
Role-Playing Games 39
Strategy Games 41
Other 2D Game Genres 42
Licensed Games 42
What Makes a 2D Game Great? 43
Interactivity 43
Game Challenges 46
The 4 Fs of Great Game Design 49
Review 51

Chapter 3 The Torque Game Builder 53
The Torque Game Builder 54
Licensing 55
Torque's Features at a Glance 56
Built-In Components of TGB 57
Other GarageGame Products 66
Torque X Engine 66
Torque Game Engine 67
The Torque Game Engine Advanced 69
Resources 69
Review 69

Chapter 4 Making Game Graphics 71
What Are Game Graphics? 72
Bits and Bytes 73
Graphics Pixels 73
Graphics Coordinates 74
Image Types 75
What Are 2D Graphics Good For? 76
Sprites 76
Tiled Images and Backgrounds 77
The Draw Order 77
Making Vector Art 78
What Is Vector Art? 78
What Is Illustrator Good For? 79
The Illustrator Environment 80

The Anatomy of Vector Objects 85
Making a Hearse 88
Using Graphics in Torque Game Builder 93
Making Static Sprites 93
Making Animated Sprites 95
Making Tilemaps 97
Review 99

Chapter 5 Programming with TorqueScript 101
Game Programming 102
Programming Languages 104
TorqueScript 104
Algorithms 106
Flowcharts 111
Torsion 112
Review 115

Chapter 6 Your First Game Project 117
Getting Started with TGB 117
Adjusting Your Workspace 118
Creating a New Project 120
Saving Your New Level 122
Loading Your Resources 122
Level Design 123
Making a Level in TGB 129
Designing Characters 137
Making an Animated Character in TGB 148
Review 172

Chapter 7 Adding Game Challenges 173
Game Challenges 173
Common Types of Game Challenges 175
Quests 176
Don't Forget to Add Conflict 178
One Lesson to Remember 178
Creating Game Challenges in TGB 180
Loading Your Resources 180
Adding a Reward 180
Providing Competition 190

Proofreading Your Program 195
Testing Your Level 197
Review 198

Chapter 8 Making a Shooter 199
Characteristics of the Shooter 199
Loading a Saved Project 200
Creating Enemies to Fight 200
Creating a Ghost Enemy 201
Programming the Ghost Enemy 201
Setting the Ghost's World Limits 204
Programming the Ghost's Spawn Function 205
Having the Ghost Destroy the Player 208
Proofreading Your Program 209
Editing the Ghost's Collision Polygon 212
Testing Your Level 214
Having the Player Fight Back 215
Giving Batty a Ray Gun 216
Creating Death Rays 217
Programming the Death Rays 218
Proofreading Your Program 220
Testing Your Level 224
Nuking the Ghosts 225
Creating a Particle Effect 225
Programming the Explosion 228
Adding Audio 228
Hollywood Sound 229
Sound Basics 231
Sound in Games 234
Using Sound Libraries 235
Mixing Your Own Audio 236
Sound in the Torque Game Builder 240
Review 243

Chapter 9 Getting Your Game Out There 245
Where to Go Now 246
The Packaging Utility 246
Learn to Advertise Yourself 249
Learn to Advertise Your Game 250

Pitch a Game Proposal to a Publisher 252
Advertise Your Game Online 253
Using Blogs or Online Communities to Get Noticed 265
What's Next? 270
Game Design Schools 270
Review 272

Appendix A Keyboard Shortcuts and Operators 273

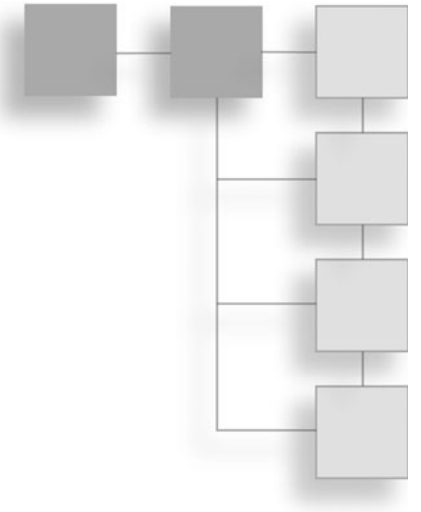
Appendix B What's on the CD? 277

Appendix C Glossary 279

Appendix D Online Resources 289

Index 293

Introduction



Welcome to *2D Game Building for Teens*! This book will help you learn about a complete computer game engine and the techniques it takes to make your video game ideas come to life.

This book is written in a tutorial format, so that as you read you don't just process information but you put it to immediate use and get hands-on learning to reinforce the knowledge. Through this book, you will start by creating a simple catch-the-coins game and end with a shooter game, but I hope you will use the skills you learn with it to springboard your talents into making dozens of computer games!

What You Will Learn from This Book

In *2D Game Building for Teens*, you'll learn about the game industry, the process by which arcade games are made, and how to make your very own games using Torque Game Builder from GarageGames. Torque Game Builder is a fantastic tool for someone just learning how to build games, and *2D Game Building for Teens* will break it down for you in easy-to-understand techniques.

Who Should Read This Book

Anyone who is interested in working in the game industry, who likes playing video games and would like to make their own, or someone who is interested in making games as a hobby and doesn't know where to start, will find the contents

of this book useful. The following text goes over the specifics of creating games with the Torque Game Builder, but it also covers the very real day-to-day responsibilities game developers have to deal with. Note: As this book is about designing computer games, you should have some experience with computers beforehand!

How This Book Is Organized

Here are some specifics about the chapters in this book.

Chapter 1: Want to Make Games?—Before delving into game production, this introductory chapter will give you a quick run-down on the industry.

Chapter 2: Brief Look at 2D Games—This chapter shows you the history, progression, and classic genres of 2D games, as well as how to think about designing your own games.

Chapter 3: The Torque Game Builder—This chapter delves into the technical information behind the powerful Torque Game Builder.

Chapter 4: Making Game Graphics—This chapter provides the information you need to design game graphics.

Chapter 5: Programming with TorqueScript—This chapter covers the intricacies of game programming and the use of the TorqueScript syntax.

Chapter 6: Your First Game Project—In this chapter, you'll learn to start a new game project in Torque.

Chapter 7: Adding Game Challenges—This chapter shows you how to set up rewards and obstacles in a game.

Chapter 8: Making a Shooter—This chapter covers creating enemies and firepower for shooter-type games.

Chapter 9: Getting Your Game Out There—This chapter delves into the future: how to get your games noticed, get into a game school, or get hired by a game company.

Appendices—Includes info about keyboard shortcuts, glossary terms, online resources, and what's on the companion CD-ROM disc.

The companion CD-ROM—The companion CD for this book contains data files used for the exercises in this book, as well as many of the tools and resources you'll need. You will find:

- **Software Demos**: Torque Game Builder version 1.74, Torsion syntax editor, and the Platformer Starter Kit.
- **Game Demos**: *Jewel Quest II*, *King Kong Skull Island Adventure*, *Magic Pearls*, *Phantasia II*, *Pirate Tales*, *Puzzle Poker*, *Rack 'Em Up Road Trip*, and *Tank Buster*.
- **Royalty Free SFX**: Great sound effects from Mojo Audio, including magic spells, explosions, sci-fi weapons, and arcade game sounds.

This page intentionally left blank

CHAPTER 1

Want to Make Games?

In this chapter, you will learn:

- The definition of games
- The ins and out of being a game designer
- How to come up with your own game ideas
- The game development production cycle
- About the independent game movement

Do you know what would make a great game? Do you have a treasured idea, a game that you wish somebody had already developed so that you could play it? Or have you ever wondered what it would truly be like to be in charge for once of making a game that you could share with not only your friends and family but also the entire world? Have you ever wanted to see a game title on the Internet with your name beside it as creator?

All this and more awaits you. You can build your very own games with little to no experience, using one of the most comprehensive 2D game engines around. *2D Game Building for Teens* guides you in using the ever-popular Torque Game Builder software (see Figure 1.1) to make your very own 2D games.

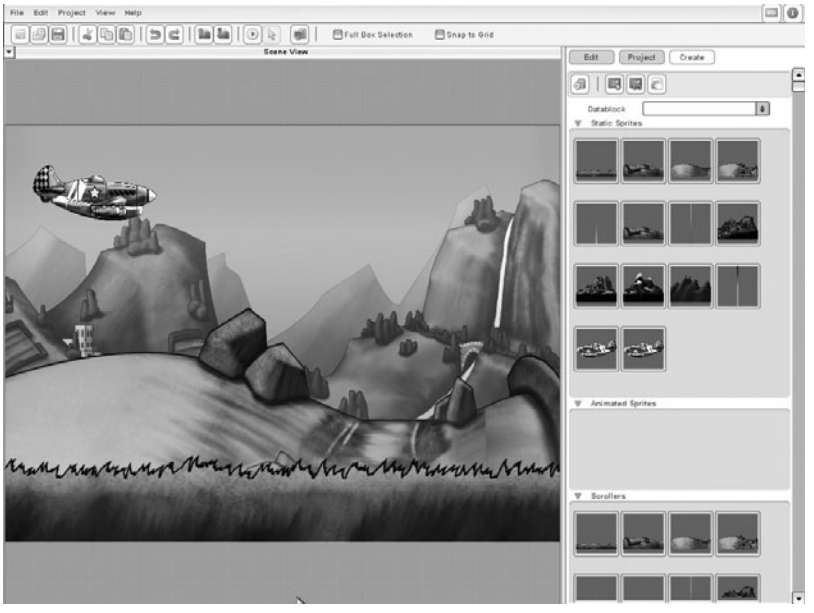


Figure 1.1
Torque Game Builder.

First, let's start by learning what it takes to be a game designer, the phases of a typical game project's development, and the importance of independent game development.

What Are Games?

Assumptions are terrible things to make, because they almost always lead to error. That's why you should know the terminology we'll be using throughout this book. Some terms you probably already know, and others may be totally new to you. Before comprehending the more difficult material later on in *2D Game Building for Teens,* you should take time to familiarize yourself with the following.

A *game* is, by simple definition, any activity conducted in a pretend reality that has a core component of play. *Play* is any grouping of recreational human activities, often centered around having fun (well, duh!). The pretend reality of most games is based on the mental capacity to create a conceptual state self-contained with its own set of rules, where the pretender can create, discard, or transform the components at will. Sounds really complicated, but actually, play is meant to be fun!

Huizinga's Magic Circle

This pretend reality is referred to by experts as Huizinga's Magic Circle, which was established by Johan Huizinga in 1971. *Huizinga's Magic Circle* is a concept stating that artificial effects appear to have importance and are bound by a set of made-up rules while inside their circle of use. For instance, the game football is about guys tossing a pigskin ball to each other, but inside Huizinga's Magic Circle, the ball and the players abide by clearly outlined rules to reach a victory for one team or the other. Consequently, the concepts of winning and losing are not essential to *all* games, but they do make a game more exciting, competitive, and positioned within a clear frame of reference.

Note

Ludology is the academic study of games for the features that are distinctly related to play, including rules and simulation. The word comes from the Latin *ludare*, meaning "to play." Ludologists are academicians who debate the finer points of gaming. You see them around colleges and universities sharing heated discussions on the entertainment value of *Halo 3*!

Core Mechanics

Electronic games have one big disadvantage to traditional board or card games: most of the game rules are hidden. The game still has its own rules, termed *core mechanics*, but they are rules that are rarely written down for the player to consult before jumping into play. Instead, video games allow players to learn the rules of the game as they play.

Harder games—or ones with entirely new/unheard-of rules—often offer players training levels in the game. These are levels of the game where players are taught the core mechanics by moderated experimentation. Given this route for learning rules, players with more practice playing a specific game will be better informed and can optimize their choices.

Hiding the rules offers video games one huge advantage over traditional games: because the computer sets the boundary of the magic circle, the player no longer has to think of the game as a game! This level of immersion is found lacking in traditional games.

Most of a game's core mechanics dwell on player interaction and randomization.

Player interaction involves a complex human-computer interface (think *The Matrix*!) where the player gives her input or feedback to the game engine and the

engine responds proportionally. In other words, the player has to tell the game what he wants to do so the game can react accordingly. This interaction can reside on mouse and keyboard or on a handheld game controller, but it comes in the form of key or button combinations and directs the course of action in a game.

Almost every game out there has some *randomization*, which is a method by which a computerized system can change the way in which a game is played. This encourages replayability in games, because the experience of playing the game is never the same twice. Randomization in classic *Dungeons and Dragons* games involved the rolling of dice, which can be simulated electronically by random number generators. However, in the digital world, randomization can go much farther than that—often where game worlds change each time the game is played, such as in the *Diablo* game from Blizzard Entertainment.

Note

Replayability is the game designer's "sweet spot." In other words, it's what every game developer hopes for and once a developer can get it firmly grasped in his game he will have a warm feeling that only comes from repeat players! Replayability in a game means the player doesn't play the game only once through but wants to play the game repeatedly, either motivated by the need to excel or by the sheer excitement that comes from experiencing a compelling narrative.

Gameplay

Gameplay is defined by game designer Dino Dini as "interaction that entertains" and by game designer Sid Meier as "a series of interesting choices." Gameplay comes first, because it's the primary source of entertainment in all video games, the storyline falling in second, and when designing a game, gameplay must be the first thing you consider.

Gameplay differs from game to game, based on the player actions, options, and challenges. The challenges are central to the game, often varying by genre, and the options are the interactive abilities open to the player in order to overcome challenges. The player actions are steps players take to achieve their goals throughout the game.

Most gameplay of the *Grand Theft Auto* game series can be condensed to: "Take missions to gain respect, steal autos to complete missions, make money by completing missions, gain respect by making as much money as possible." See Figure 1.2. The speed at which these actions are taken makes up the *pace* (also called the *game flow*) of the electronic game.

Figure 1.2
Games like *Grand Theft Auto* aid in catharsis.

Tip

"Like a film, a game is a powerfully dramatic medium—one that requires ideas and vision in order to move the audience, whether they are players or viewers, emotionally and intellectually. Growing up, we are not really taught how to communicate ideas using dynamic systems in the same way that we are taught to write essays or tell stories. But there is a lot we understand about the world that can't be told using linear media. Game systems offer us a new way of making stories and communicating this type of knowledge."

—Tracy Fullerton, Assistant Professor at the USC School of Cinema-Television

Industry celebrity Michael E. Moore wrote that there were specific gameplay elements, including the following, inherent in all games, and that you could rate a game's entertainment value based on percentages of each:

- Combat
- Construction or destruction
- Driving or piloting a vehicle

- Exploitation
- Exploration
- Physical dexterity
- Puzzle solving
- Storytelling

If you want to rate your game, or any game for that matter, by means of the above elements, just divide 100% among the individual traits the game displays. As a case in point, a game without any puzzle solving in it would rate 0%, while the classic arcade favorite *Mortal Kombat* would probably rate close to 100% on the combat.

One of the thorniest facets of creating a successful game is making sure it has balanced gameplay. This means that no one specific feature of the game outweighs another. If the player learns that his crossbow beats all monsters, possibly by some programming fluke, she'll never try another weapon during the course of the game—not even when you try giving her a weapon she'll need to complete the game later on!

Players try to find the easiest and most efficient way to beat any game, because games have an underlying competitive challenge, even when the only competition a player faces in a game is the computer brain itself. So be on the lookout for minor imbalances in gameplay and the core mechanics and repair them so they are more reasonable.

When you think you've found all the discrepancies, have a friend or two play it with fresh eyes and see what they discover. They might find a loophole or problem you missed.

Graphics

There has long been a debate over which is more important, gameplay or graphics, in video games. In truth, both are of equal value and should be treated as such.

In the early days of arcade games, the weakness of the display hardware seriously hampered the aesthetics, resulting in ugly and oversimplified graphics. With the growth of modern display technology, graphics have taken on a much greater

role, one that some designers see as a handicap. In the 1990s, there was a major push by Hollywood film production companies to take over the game industry, and to a certain extent because of this thrusting interest, game companies became more focused on their outward appearance.

Part of your job description as a game designer is to give players aesthetic entertainment. Any ugly or awkward game with poor artistic style, clumsy animation, and sloppy artwork will disappoint gamers, but the appeal for games lies just as heavily in consistently fun and innovative gameplay.

On the flip side of this, however, you don't have to be a great artist to make fun games. There are countless amateur-made games featuring stick figures out there. The idea is to make a fun game, and if your game is enjoyable, the graphics become window dressing. So if you aren't confident in your abilities as an artist, you can (a) take art classes and get better at it, (b) bribe your nearest artsy friend into helping you, or (c) do your best and move on.

How to Be a Game Designer

The game industry hasn't been around that long. It spawned from hacking communities in the early 1960s, and most of the industry veterans haven't even reached the age of retirement yet.

These days, being a game designer is considered the same as being a rock star was several years ago. Game designers are seen as young and influential guys and gals sitting on the cutting edge of the wild side, using computer technology to create electronic games for the masses. Game designers literally make fun! So why shouldn't their job be viewed as one of the most fun possible career opportunities ever?

Not only does their job have the added perk of being fun, but there's money to be made at it! In 2007, this relatively new field brooked $18.85 billion, and sales of video games topped sales of CDs and DVDs—making more money than either of its competing entertainment industries: motion pictures or music. In fact, data from the NPD Group and announced by the Entertainment Software Association (ESA) reveals that first-day sales of Bungie's title *Halo 3* outsold first-day sales of the final book in J.K. Rowling's popular Harry Potter series! Master Chief says, "Beat that, Dumbledore!" See Figure 1.3.

Electronic games are spanning the earth and this global expansion has ushered in a need for skilled programmers and talented game artists. Most tech schools have

Figure 1.3
Halo 3 beat *Harry Potter and the Deathly Hallows* in a run for first-day sales.

started offering degree programs in game design to help fill this need, creating a brand new foundation in game education. (The most prominent of these schools will be discussed in further detail in Chapter 9, "Getting Your Game Out There.")

A single triple-A game title, like Bungie's *Halo 3*, may have from 50 to 200 talented individuals on its project team—and generally costs the same as a Hollywood blockbuster to make. Because it costs so much to develop a triple-A title, most development teams are funded by a big-time game publisher such as Microsoft, Nintendo, Activision, or Electronic Arts. Though this funding assists in reducing the enormous costs of building a game from scratch, the publisher gains exclusive publishing rights to the finished game and the game must be set to make money before its release date, often a daunting challenge for a new game developer.

Note

When we talk about a *triple-A (AAA) game title*, this description refers to an individual title's success or anticipated success if it is still in development. Triple-A titles are defined by the cost spent and the return-on-investment. Most triple-A title games cost between $10 and $12 million to make and become a smash hit, selling over a million copies.

Game designers fall into several skill-based job classifications. The broadest classifications of game designers include:

- **Artists**—create the game's major thematic style through the development of concept artwork, 2D characters, 3D polygonal models, and other visual assets, including props, weapons, vehicles, and monsters.

- **Leaders**—communicate between the team members and make sure everyone is doing what they should and that development milestones are reached on time.
- **Level Artists**—take the design documents and the art bibles and use apparatus therein to construct the individual maps, levels, and environments players will play through during the course of the game.
- **Programmers**—make the most money, because they have to script the program code that lets the computer know what to do and how to react to the game's players.
- **Sound Engineers**—set up Foley sound effects, musical scores, ambient sounds, and voiceover narratives to make the game sound so sweet to listen to.
- **User Interface Artists**—design the look and feel of the game's shell interface, including the menu screens and in-game options lists. UI artists must test usability against aesthetics to maintain thematic style.
- **Writers**—not only have to write the storyline for the game, but they also script the dialogue and events that take place in the game, and write the game manual.

Make a note that the jobs and descriptions listed above are vague enough that several roles fall under each one. Consider them umbrella categories. Also, most game companies list types of jobs under different titles even though they have the same job description (i.e., you'll see level artists also called world builders, world designers, environment artists, terrain artists, terrain builders, and more). Remember, too, that this is a relatively new industry and one with lots of room for experimentation and exploration. There aren't any concrete rules.

This book should help you in becoming an amateur artist, level artist, programmer, or leader. You have to learn to crawl before you can fly, and the exercises herein will benefit you in doing so. Use it wisely and you can excel and some day find yourself flying along in the fast-paced game industry.

Tip

"Making games fun is the most challenging aspect of the game-development process. Everything else pales in comparison. Games have to be familiar enough so that they are accessible to the audience, but different enough to be novel and challenging. It's such a balancing act when

making a game—to find and iterate through game systems, be strongly self-critical of features that aren't adding to the fun of the game, and still be able to defend those that just need a little bit more work to gel into something great."

—Mark Terrano, Technical Game Manager for Xbox Advanced Technology Group

You do not have to have a formal education to become a computer-age rock star. In fact, employers, recruiters, and headhunters typically look at your *portfolio*, or a list of what you have done successfully, when they are eager to hire you. Thus, it is imperative for you to start early making games. If you have one or more game titles that you have built yourself inside your portfolio, you have a much better chance of attracting the eye of a game development company!

Let's take a look at one game development company, Foundation 9. Of its more than 450 employees before its merger with Amaze Entertainment, Foundation 9 had approximately 20% designers, 30% programmers, 35% artists, and 15% producers, administrators, and executives. Many of its employees "cross-pollinated"—in other words, some programmers became designers, and some designers became producers. The average development cycle at Foundation 9 was 14 months, and project budgets ranged from $1.5 to $12 million (or more) per title. They never take breaks, either: Foundation 9 ships between 20 and 30 games every year!

There are some very important traits shared by all game designers. First of all is the passion. If you don't have a passion for games, you will get bored working on them easily and, due to the extreme focus it takes to build games, you'll get burnt out faster.

So ask yourself how often you play games and whether you love games enough that you want to make your own and don't mind how much work is involved to do so. If you still want to be a game designer, then this book will test your mettle, so to speak, and help you see if you have what it takes.

The following are basic skills useful to most game designers:

- A huge imagination
- Technical awareness, especially computer knowledge
- Analytical skill (the ability to recognize the good from the bad)
- Math skills

- Artistic talent
- Communication skills
- Research skills
- Mental focus and the ability to get things done
- The capability to adapt fast and compromise on major issues

Noah Falstein, president of The Inspiracy and well-known icon of the games business (see Figure 1.4), says, ''Game designers are universally fascinated by what makes people tick, what makes *everything* tick. For the most part, they are somewhat introverted, although there are some who are extroverts. But that degree of introversion is necessary to sit and noodle out how a design is going to work. In game design that introversion and extroversion tends to balance a bit. The personality of game designers is remarkably consistent, and this is an interesting point, specifically with full-time lead-designer types. There's a sense of kinship we have for each other in this industry.''

Figure 1.4
Game biz icon Noah Falstein.

Tip

"As lead designer, the actual duties vary on a day to day basis. Overall, I'm responsible for keeping the vision for the game, the game mechanics, and the 'fun' of the game; the overall story (and any specific elements about the game designed to propel the overall story, such as companions, key locations, etc.); and then breaking down the remaining elements into digestible chunks for the other designers in terms of area briefs and area overviews . . . breaking up the mechanics and play-balancing . . . and then managing all the parts so programmers, artists, and the producer are getting everything they need to keep moving."

—Chris Avellone, Lead Designer at Obsidian Entertainment

Come Up with Your Own Game Ideas

Although this book gives you exercises to introduce to you the tools and techniques you'll need to make your own games, don't stop there.

Game ideas can come from anywhere, including other games, TV shows, movies, and alternative media. But you can't outright steal another person's intellectual property. You can't make a game based on *Pirates of the Caribbean* without stepping on toes at Disney, but you could make a fun cheerful game about pirates, such as Twintale Entertainment's *Pirate Tales* game (as you see in Figure 1.5).

CEO of Online Alchemy and multiplayer online game guru Mike Sellers says, "Ideas are very much the easy part. I literally have a file drawer full of game ideas, most of which I'll never get to. Ideas are important, but just really the first step, or the zeroth step. Once you have that seed, you need to go back to how does the player feel, what's their journey, or what's the task, what are they trying to accomplish—the nouns and verbs."

You probably already have great ideas for games, but if you're not like Mike Sellers or are uncertain how to come up with your very own ideas, here are some tricks to brainstorm them:

- Ask your closest friends what games they would like to play.
- Watch a movie or read a book. Think about the possibilities. Could this non-interactive work be adapted into a game? What game genre would it be? How would you play it? Because these are licensed properties, you can't hope to make money off borrowing from other people's creative ideas, but this is a great jump-off step.
- Choose one of your all-time-favorite games from the past and clone it using new technology. Don't try to make money off copying someone else's

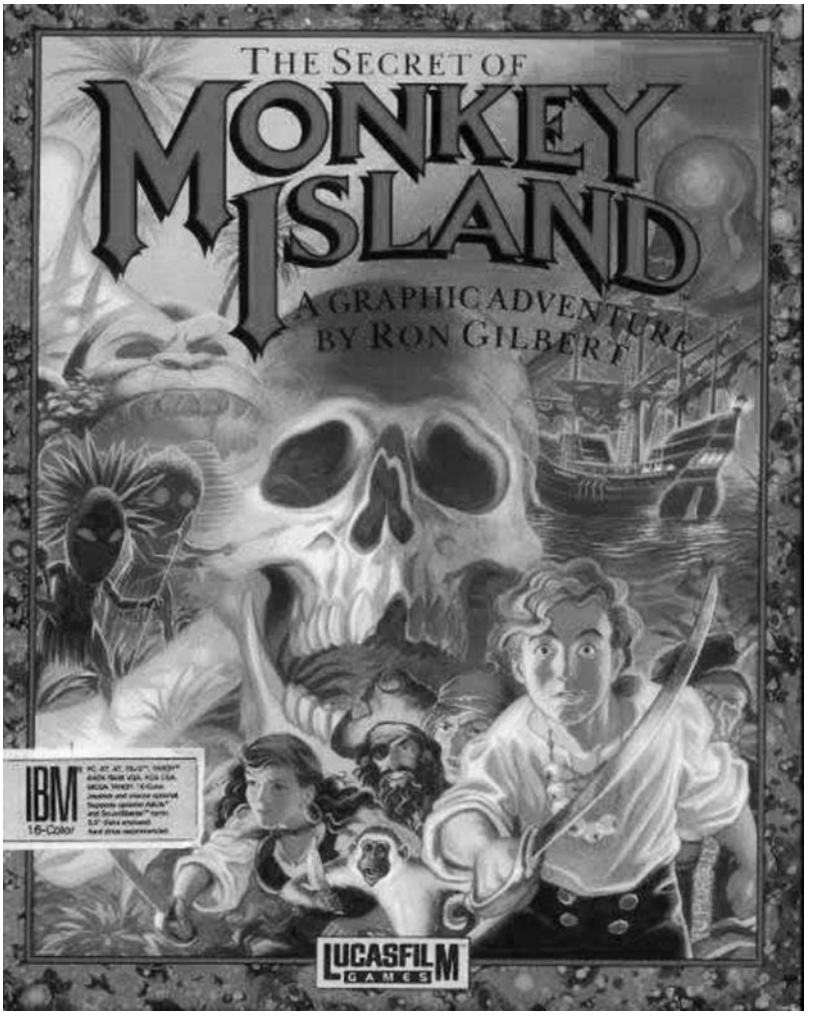


Figure 1.5
The *Pirate Tales* game from Twintale Entertainment.

creative work, however. Do this only for practice or to get recognition within the independent gaming community.

- Play any game you like and analyze it. Pick what you like about it, what you don't like about it, and what the designers could have done better. Take notes, and use those notes to make your own original work.

You can find game ideas almost anywhere but only if you're looking for them. Creativity is an active, not a passive, process. Be ready for it when you are struck

by inspiration and have a notebook at hand to record the genesis. Also, remember there is no such thing as a new idea. Most ideas out there have been done repetitively before. Take an existing idea and make it better. Put your own spin on it.

Tracy Fullerton is the assistant professor at USC's School of Cinematic Arts, Interactive Media department, and co-director of the Electronic Arts Game Innovations Lab. She says, ''Brainstorming is different for every project I've worked on. I've seen anything from 10 to 11 people, half of whom are clients, sitting in a room, to sitting alone at home and brainstorming ideas. I don't prefer either way. The most important thing is that you have a goal, you have enough information to work from, you've done enough research where your research is informed, you have an understanding of those problems before you, and you have a creative understanding of what your needs are.''

She also understands the need for creativity and having fun when brainstorming. ''Some of the best brainstormers are Imagineers [the kind folks who work at Disney Imagineering]. They often have very large brainstorming sessions, with people from very different backgrounds, and they have physical toys to keep people loose. And somehow, tossing toys around gets the creative ideas flowing, and I find that very successful.''

This process sounds wild and crazy, but it's actually encouraged in lots of companies that focus on creative entertainment. The more invigorating executives can make the work environment, the more productive the staff will be.

Noah Falstein, mentioned above, replies to a question about brainstorming: ''One example you hear about all the time is, 'Don't criticize; don't stifle the process by judging; every idea is a good idea.' Under corporate circumstances, that might be useful. But all hardcore game designers criticize each other left and right. The difference is they don't take it personally. It's okay to critique. That way only the strongest elements survive.'' He adds, ''Darwinism is very appropriate to correlate with game design.'' When he brings up Darwinism, he's talking about natural selection, or the theory that only the fittest elements will survive to procreate. In idea genesis and game development, there's a direct correlation. If game elements don't serve the purpose of the designer, or the programming of one tiny detail is taking too long or becoming too intense, then the element is tossed out and only the best elements remain.

Daily conversation with fellow game designers through a creative process is not stifling at all. Studio professionals at Blizzard Entertainment in Irvine, California, often sip Starbucks coffees while sitting around in bean bag chairs and sketching on whiteboards during their brainstorming process.

You'd practically pay to work in such freethinking and fun environments! This is one of the many perks for becoming a game designer. However, it's not all fun and games in the land of make-believe. Once you come up with your own wild ideas, you are going to have to write them down.

Write Your Ideas Down

Once you have an idea, it's time to put it into words. Expressly you should write a game concept encapsulating your game idea. A *game concept* is a short description of a game detailed enough to start discussing it as a potential project. The concept forms a general idea of how you intend to entertain someone through gameplay, and, more importantly, why you believe it will be a rich, compelling experience.

First, take stock of who your audience will be. In a commercial environment, publishers define this audience as a *target market*, or a specific group of people to sell to. In a player-centric design, you put the players first and design the game around their expectations. You could design a game for yourself and hope that there are a lot of people out there who are just like you who will find the game entertaining, but it has been proven that you will make more return on your initial investment of time and money if you publish a game for a specific group of gamers instead. So look around at what types of gamers you see and who you think would like your game. Comparing your work to previously popular game titles is one way of doing this research.

During this step, try for inclusiveness and not universality; what I mean is, don't try to make a game that will appeal to *everyone*—just make a game that will appeal to the largest segment of gamers.

Keep in mind that there are *core gamers*, who routinely play lots of games and play for the thrill of beating games, and there are *casual gamers*, who play for the sheer satisfaction of the experience and are less intense about the games they play.

If you put a secret reward in your game, often referred to as an *Easter egg*, it might escape most gamers' notice, but you can expect a large number of core gamers

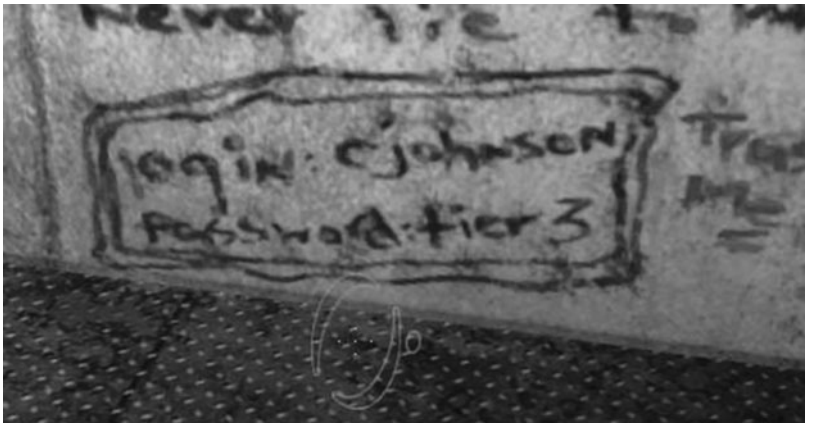


Figure 1.6
An Easter egg found scrawled on a wall in Valve's game *Portal*.

will discover it. Remember to design your game for both types of gamers. One example of this is Figure 1.6; Valve's experimental game *Portal* had a cult following and a website called http://www.aperturescience.com. Players who discovered the Easter egg in the figure could login to the site and uncover fun and informative secrets to the game.

Secondly, determine the player's role in your game. In an abstract puzzle game like *Bejeweled*, the player doesn't actually become immersed in a fictional game world because she doesn't really have a role in it. But in a representational game, the player does a lot more pretending. She pretends there is a game world, she pretends to be in control of a particular character's actions in that world, and she reacts to the situations you put her in based on that role-play.

In such games, the player has a *role*. You must be able to put in plain words the player's role—and that role's responsibilities—clearly. Leave little doubt what their mission will be in the game.

Note

Concept design is critical, and one of the problems I want to warn you about is the danger of irresolution. Too many designers face indecision when it comes to pouring the foundations of their games. They say they're "keeping their options open" and are fearful of making bad choices or overlooking something. However, as the game switches into full fabrication, if the most critical facts about the game are still jerking around or foggy in nature, production will become sluggish and inefficient because of a lack of visualization and confidence. Projects sink quickly. So don't waver about making design decisions. Make the decisions now and revise later as needed.

Next, write the game concept down on paper. As a minimum, your game concept should include:

- A *high concept statement* that is a two-to-three sentence description of your game. Remember those blurbs of upcoming movies seen in the *TV Guide*? Keep your statement short and sweet.
- Who the player will be in the game, if they have a character they can see onscreen.
- What the game world is like, if there is one.
- The game genre or a clarification of why its gameplay doesn't fit any known genre.
- The projected gameplay mode, including camera perspective, interaction, and gamer challenges.
- Who the target audience of the game will be, including their demographic (age, gender, how much money they bring in, and more).
- The type of machine and input devices the game will run on.
- Property licenses the game will exploit, if any.
- A short summary of how the game will progress from level to level, including a synopsis of the storyline.

Here are a couple of examples of game concepts:

Game Concept 1—Jake Space Racer

Take off into the wilds of outer space as a rocket shooter, and make big galactic bucks in a daring game avoiding asteroids and creepy bug-eyed monsters.

The player is Jake, a dopey blond buckwheat in cowboy boots who rides a red rocket that looks like it was welded together from scrap metal. He speaks in drawling metaphors, like "It's hotter in this nebula than the business side of a skillet!"

The game world is an oversimplified cartoon universe that is full of commercial ventures in outer space. Asteroids are anchor posts for billboards, diners float by

on antigravity parking lots, and every part of space is populated with weird, cartoony caricatures of extraterrestrials.

This game is a racing game, under the action genre. It requires fast reflexes and sharp eyes to avoid obstacles on a timed course event. It will be played in third-person camera view with a graphic user interface showing the player how many galactic bucks they've won so far, the timer once the race has started, and what place the player is currently in the race. Obstacles during the race include other racers, controlled by the computer, dirty alien saboteurs who appear in low-flying UFOs, exploding mines, and gravity-sucking asteroids. Racing courses are loopy, physics-defying, and long—like insane Moebius strips.

Players will most likely be casual gamers, fans of racing games, or fans of science fiction themes. They will be more than 50% male and young, between ages 7 and 35 years old.

Play will progress level by level for 40 levels, starting on planet Earth and ending up at the Milky Way Championships, where the player will compete for a space racing trophy.

Game Concept 2—Jenny and the Haunted Schoolhouse

Solve the mystery of the disappearing Woman in White at the schoolhouse on the hill, starring intrepid amateur sleuth Jenny Scorcher.

The player is Jenny Scorcher, a small girl with large glasses and a perpetual raised eyebrow. No matter what the situation, her rampant curiosity gets the better of her common sense—making her intrepid and unafraid of going where others fear to tread.

The game world is the Gaitswater School for Preppy Girls, a scary autumnal house perched atop a gray hill, surrounded by looming leafless trees. The exterior of the schoolhouse is brooding and unkempt, because the school has been closed for years with only a singular swarthy caretaker to maintain the grounds. The interior has seen better days, as its Victorian architecture is swathed in dust and cobwebs and its antique furniture is draped in white sheets.

This game is an adventure game, featuring puzzles the player must solve in order to reach the further depths of the Gaitswater School for Preppy Girls in an effort to find out where the mysterious Woman in White went. It will be played in adventure scenes, with the player maneuvering a 2D character from one scene to

the other. There will also be a graphic user interface showing the player's inventory items and Jenny's clue book, which can give the player clues as to how to proceed if they are lost. Obstacles include puzzles, which can take the form of mini-games, such as color combinations and Rube Goldberg machinations.

Players will most likely be casual gamers, fans of adventure games, and fans of mystery stories. Over 50% will be female and between the ages of 5 and 30 years old.

Play will progress from one adventure scene to the next until Jenny (and the player) discover the truth of the Woman in White's whereabouts.

Game Design Step-by-Step

There are many steps to designing games. When you play a finished game you don't see the manpower it took to polish the game into the fine piece of electronic make-believe you sit down to play. Some games take years (and tears!) to make.

The steps can be broken down into three broad processes:

- Preproduction
- Production
- Postproduction

Preproduction

The *preproduction phase* of game design takes place before designers ever get their tools out and get started. A game idea is finalized into a working concept and the core team of creators starts fleshing out the game:

- Artists come up with an *art bible* of all related concept artwork, including drawings of the characters, vehicles, environments, and weapons.
- Writers pen the original *design document*, which tells the team all the details of the game, including which levels and characters will be in the game and how the player controls will work.
- A short playable demo—what is often referred to as a *prototype*—is thrown together to be a proof-in-concept.
- Then the leader works with his team to prepare a *game proposal* that is sure to knock the socks off of prospective investors or publishers.

Depending on the size of the game project, development teams will allot anywhere from one to six months of pure preproduction work before starting production.

Design Documents

Beginning game designers like to come up with an idea for a game and plunge into programming without preamble, but this sort of ad hoc approach is disastrous.

No artist can work in a vacuum. No one who's considered a professional in the industry would condone going right from the idea stage into the coding stage without a proper pencil-to-paper design work up front. It's a whole lot easier to build a game when you have a blueprint to go by, and that's the number one reason design documents are important to write.

If you ask an artist if he is done with a painting he's working on, he'd give you a funny look and tell you, "Never... I'm never done." Most artists are never entirely finished with a work in progress. They continuously edit and modify their pieces to fit their ongoing vision. In game design, this is not recommended. You want to get your finished work done and put out there for the masses to play, right? You might also count on publishing your work to make money off it to pay yourself back for all the hard work and investments you've put into it. So you want to cut to the chase, get the work done, and when it's through you want to know it's completed. For this, game designers operating in a team environment must make sure they're getting done what they need to and not moving backward or wandering off on tangents.

Presumably, a development team uses the documentation to steer every step of game creation and to stay in touch on what each member is doing or needs to do. However, in real life this is what daily team meetings and lines of communication do. The design document, however, *does* create a nice paper trail for the production, and it nails down potential problem areas before the team gets to them. It also helps set the tone for the final product.

It's often difficult to make a game that looks as fresh as the initial idea. In other words, you want to make your game look great the first time around and when you repeatedly work on it, developing it through multiple iterations, it loses that fresh "never-been-done-before" look. One of the primary habits for staying consistent with your early vision is to stick closely to the game design document and to have final product goals clearly and concisely listed therein.

Note

Most design documents not only list product goals but also a development schedule. The producer, or project manager, works closely with the lead programmer, artist, and designer to apply a timeline that lists all tasks that are needed to complete the game and realistic steps of production that can be accomplished on that timeline. These steps are called *milestones*.

No two design documents are alike, and some vary based on intention of purpose. Here are examples:

- A *high concept document* highlights the key elements of the game in bite-size chunks for the purposes of grabbing producer or investor interest. It is not used to develop from, because it does not give enough information about the game.
- A *game treatment document* is almost like a brochure, summing up the core ideas behind the game but with more technical gameplay details than the high concept will have.
- A *character design document* specifically targets one character, often the avatar of the game, and describes that character's appearance and *moveset*, or the list of animations documenting the character's movements.
- The *world design document* describes the background information and the sorts of things that the make-believe world will contain in a general overview. Most world design docs include a map of the world used, along with a list of objects and ambient sounds and their appropriate locations.
- *Story and level progression documents* record the storyline and the way that the levels will progress from one to the next. If you're creating a small game that won't have levels (like a Poker game) or use a game genre without a story (like a puzzle game) then this type of doc would be superfluous. Otherwise, you should write one out and in it indicate just how the player will experience and interact with the story, including the gameplay options and the narrative devices used.
- Once upon a time, a single game design doc was written for each project, called the game script, but it was huge and unmanageable—one of the reasons that it got sliced into separate category-specific docs. The *game script* is used for a complete overview and for one purpose not covered by the others: to target the key rules and core mechanics of gameplay. The

theory behind playing the game should be present and enough details given that the reader could potentially play the game in their head.

- Another type of design doc, the *technical design document*, is created by the lead programmer on the project, using the game script as a springboard to describe to the technical staff how the game should be built or implemented in the software.

Chris Ferriter has over 13 years of game and motion picture production experience. He has produced and designed games for THQ, Midway, Electronic Arts, and Ubisoft. Chris Ferriter has this to say about design documentation: "When I was designing, I would try to make docs as visual as possible . . . It's very common for design docs to be created and shelved and never used again because they're big and unwieldy and not always kept up-to-date. So six months into a production when programming has a question about how a game mechanic is supposed to work, they're unlikely to go back to that doc and find the one paragraph to describe that. Instead, what they're going to do is go to the designer and have a meeting about it, and the designer will have to convey how this works. But this is lost efficiency. So what I would try to do is make a document that people will be likely to go back to. Make a document that's easy to read and easy to pull out pertinent information from."

Although the majority of companies still use pencil-to-paper documentation, many are changing with the times and utilizing documentation organized in a Wiki format live online with major topics like design, art, code, and so forth. This makes the design document more fluid and flexible while remaining informative. If you are interested in creating a Wiki style of game design document for your game, you might try the free Wiki services of PBwiki online at http://pbwiki.com. Wikis can provide free, secure means for the entire development team and producers to stay on top of the development process and access anywhere in the world.

Prototypes

In game programming, there's some uncertainty as to whether a new idea will actually do what is desired. New game projects often have unexpected problems. Plus, building the full design is often expensive and can be time-consuming, especially when designing a triple-A title, and so a lot of preproduction work is put into developing a prototype to see if the design would even work before the artists and programmers get to work.

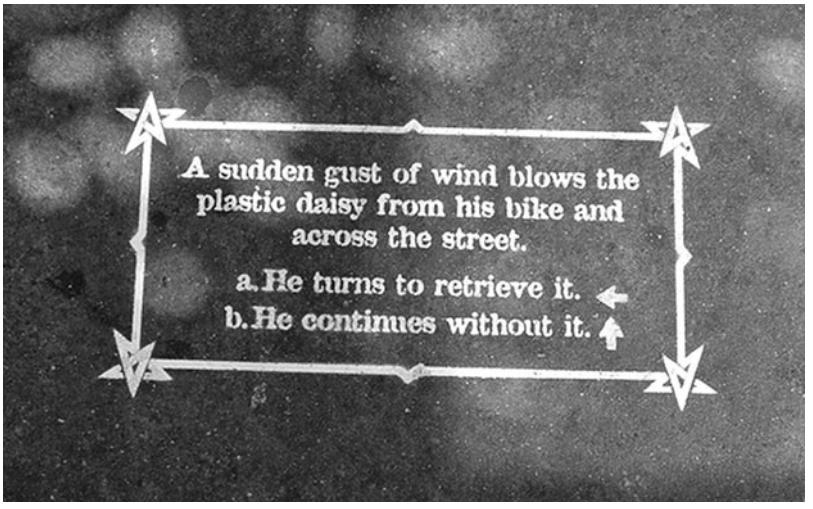


Figure 1.7
The Mission Stencil Story is a *Choose Your Own Adventure* style game played through graffiti written on the sidewalks of the Mission district of San Francisco, with lots of branching plotlines and dead ends.

A *prototype* is thus built to test out the functionality and feel of the new design before starting production of the actual game. The construction of a fully working prototype, the ultimate proof of concept, is the designer's final check for flaws in the thinking and allows last-minute improvements to be made before larger production starts.

Some prototypes are simple and resemble text-based games like the old *Choose Your Own Adventure* game books, with core mechanics that the final game will have (see Figure 1.7). Some prototypes are slapped-together board or card games made out of paper and cardboard that test out player character, enemy placement, and gameplay. Still other prototypes are designed in an easier or faster format. Some are built on the beta version of the game's engine, with stick figures for characters.

No matter what prototyping is done, even if it's done only in your own head, it will assist you in making important decisions before putting fingers to the keyboard.

Production

In the industry, you hear the term green light a lot. When a game project has reached *green light*, that means that the development group has completed the preproduction phase, the required tools and finances to begin proper game

creation have been acquired, and that the team is now geared up to start development in earnest. During this *production phase*, the artists design the assets, characters, and environments on their computers, the writers set out dialogue and scripted events, the programmers code the controls and character behaviors, and the leaders make sure no one walks off the job.

This production process often starts off wonderfully and becomes more intensive the closer to deadlines time gets, resulting in overtime during this *crunch time*. Team members may work obscene hours all through crunch time, even sleeping underneath their desks! There have even been debates about ''quality of life'' from the Electronic Software Association (ESA) since some designers' family and friends complained about not seeing them during crunch time and the designers themselves not getting adequate compensation for the additional hours they put into production.

Production does involve a unique type of testing crucial to the production phase, called *rapid iterative development*. Designers test ideas daily, see what's working and what's not, and abandon hurdles that are too difficult to get over. If the lead designer wants to see giant bumblebee enemies in one section of the game, the programmers will see if they can program flight abilities for the enemies. If the programmers can't get the bumblebee enemies to fly right, the team will discuss compromises. Also, programming is often a series of iterative stages. In the above example, the engineers will first try to get the bumblebees into the air and then see if they can get them to move around, and there will always be instances where they have to test, then revise, then test again.

Postproduction

After the game is finished, it's still not complete! That's right. Testing, quality assurance, and bug-fixing are initiated for the duration of the *postproduction phase*, followed by a public relations campaign to get the game noticed by its target audience.

Testing is often done iteratively to ensure there are no glaring mistakes; this means that testing is done every step along the way and after a mistake is fixed the game is tested again to make sure the fix didn't break something else. After the entire team has approved of their own game, they'll often pull in people not related to the team to test the game with a set of fresh eyes; this is known as a *beta test*. Even after the game releases and it is sitting on store shelves, more bug fixes or patches may be required.

Quality assurance, or QA, is not the same as testing. QA looks at the game as a whole and tests it against the initial concept for consistency.

Note

When Blizzard Entertainment launched their massive multiplayer online role-playing game (MMORPG) *World of WarCraft* on November 23, 2004, the game swiftly sold about 240,000 copies internationally. The servers they had placed the game on quickly became unsound, and Blizzard Entertainment was forced to stop selling their games in retail outlets until they could get more servers up and running and troubles fixed. They finally did get it all worked out and now have sold well over 3.5 million units of the game and have an average of 500,000 subscribers playing it all times of the day.

The *gold master* is the final edition of the game with all the bugs removed (hopefully). The phrase most likely resulted from the music industry and refers to the master disks sent to the manufacturer. These disks had to be first-rate and of more undeviating quality because they were used as the basis for the mass-produced retail copies, so they were made of gold. After the gold master is concluded and tested, it is sent to the CD duplicator to be reproduced for mass distribution.

Most of the postproduction phase of a game deals chiefly with marketing and distribution of the game. Because the game has to start making money before it's even released, the public relations person at a game publisher or developer must make sure people know about the game early on. Game magazines feature previews of early prototypes of the game or interviews with the game's creators. Websites and forums are also great places to lead on curious target audiences.

Tip

"The most challenging part of game design is not letting go of a pet idea because it's not technically feasible—or even trying out a game idea that simply does not work and having to start over. No, I think the most challenging part is the time involved. You frequently don't see a game on a shelf for at least two years (often more) after you've begun. The process has an extremely extended gestation period. But seeing it all come to life is incredibly rewarding."

—Patricia Pizer, Freelance Game Design Consultant and Founder of the Boston Area Game Developers' Network

Think about it. Most large retail outlets, chief among them being Wal-Mart, K-Mart, and Target stores, allocate a restricted amount of shelf space for games. Most of the stores stock between 200 and 300 titles at a single time. In a representative year, some 1,500 titles might get released. The retailer is thus pressured by supply and demand to get rid of titles that don't sell immediately. As a result, the typical shelf life of a game can be anywhere from 2 weeks to 6 months. Most

games make their best money in the initial 60 to 90 days post-release. Many may quickly wind up in the bargain bin or returned to the manufacturer if sales don't carry on strong. This is why adequate promotion and distribution of a game is imperative to the success of a game, and some may argue that it's even as important as the content of the game!

Note

There is a great online community for independent gamers, developers, and publishers, and it should be your first stop when you're ready to get your game noticed! It's called the *Great Games Experiment*, or GGE, and was originally created by Jeff Tunnel, one of the founders of GarageGames. GGE has built-in widgets, groups, forums, and places to upload and play games of all kinds. Loads of newbie game developers go there to get their work noticed, so check it out on the web at: http://www.greatgamesexperiment.com.

Starting Your Own Game Company

It is more complicated to start up a new game company now than it used to be. Game development needs top-of-the-line equipment and software, which doesn't come cheap if you want to do the job right, and it requires a hefty team of talented folks who will likely spend the better part of numerous years working on a single assignment. To start a computer or console game company in this way would probably run you a capital investment of about $5 to $15 million.

Of course, a relatively smaller group of impassioned gamers can make their own games for quite a bit less. By concentrating on initially doing smaller projects with premade game engines (such as the Torque Game Builder!) or targeting slighter markets, a group can develop a reputation for creating value games. With this sort of reputation in the making, the group can take on more projects and hire on more talented people until they have the portfolio and the staff that triple-A title producers look for.

The Importance of Being Independent

Tip

"I believe that the online distribution market for games is where most independent developers can get started and make a good living. Developing a game in four to six months with a small remote team means that your costs stay low and your return on investment does not have to be as large as the box or console industry... You don't have to just build puzzle games anymore to succeed in online game distribution. Many companies have proven that true—with games like *Orbz*, *Marble Blast*, and *Tennis Critters* (all available from GarageGames)."

—Justin Mette, President of 21-6 Productions

Just as game designers are the rock stars of today, the game industry is following the direction of the music industry fairly closely.

Music is big business. Typically record companies compete for listing in the Top 40 and music artists whine when they're not getting the respect they think that they ought to be getting. However, in the music industry there are still some artists that are not afraid to experiment and create really edgy tunes outside the mainstream. From garage bands and unlicensed artists comes the wild side of independent music. Indie musicians are sonic artists who aren't afraid to take risks. They settle for lesser gigs so they can play the music they want to without the heavy influence of record companies.

In precisely the same way, indie game designers are also rebels who thumb their noses at the big industry giants. If you want to see real innovation in the game industry, you have to peer at the margins, at the indie game designers.

GarageGames considers any game designer or game company that earns less than $250,000 per year to be an independent. I truly doubt if you're reading this book that you make that much right now. This means you will be considered an indie game designer and have rights to the Indie License of their Torque products, guaranteeing you a substantial discount.

Several developers, including Manifesto Games' very own Greg Costikyan, got together in the summer of 2000 and wrote the Scratchware Manifesto, which is a statement of purpose calling for game designers to stop paying attention to last year's A-lists and start developing novel experiences right now. *Scratchware*, a term coined at the time of the writing of the Scratchware Manifesto, is any game that is completely original, has great gameplay and replayability, runs well with few glitches, costs consumers less than $25 to purchase, and was created by less than three people.

Many indie games fit the criteria of scratchware. They are often shorter games, made by a handful of people in their basement or garage, and sell dirt cheap over the Internet. Just giving away your game doesn't sound very bright, and when you see prices at $10 a purchase you might wonder if indie game designers ever make their money back—yet most of these developers sell at least 4,000 copies in their first year, and at $10 a copy, that comes to $40,000 in gross profits!

Plus, as an indie developer, you can make any game you want. Because your hands are not tied by anyone else's money, you can try radical new things no

one's ever tried before. Create your very own dream game and sell it over the Internet without having to split the proceeds with anybody!

2D Game Building for Teens and the GarageGames' Torque Game Builder software give you the tools you need to start making your own computer games and get into business as a game designer—right now!

Tip

"I think the most challenging aspect of game development is just the raw creativity that needs to go into giving life to characters, the world, quests, and events. After that, everything is just grunt work that you can muscle through. Oh, and begging for resources. And selling someone on the idea. Then threatening them. Then begging again."

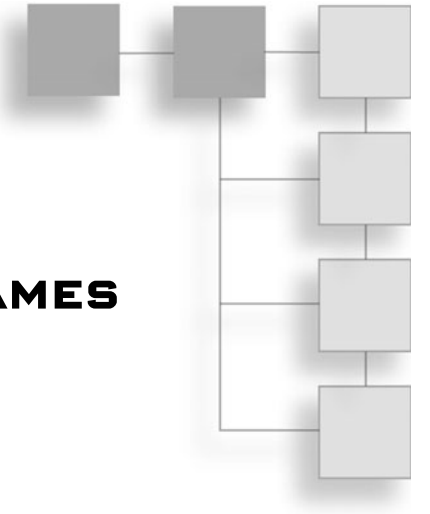
—Chris Avellone, Lead Designer at Obsidian Entertainment

Review

After reading this chapter, you should understand the following:

- How to become a game designer
- How to come up with your own game ideas
- What the game development production cycle consists of (preproduction, production, and postproduction steps)
- Why the independent game development scene is so important

CHAPTER 2



Brief Look at 2D Games

In this chapter, you will learn:

- The history of electronic games
- Some electronic game firsts
- The difference between 2D and 3D game graphics
- About the most popular 2D game genres
- Interactivity (empowering the player, immersion, and emotioneering)
- How to create gamer challenges
- The 4 Fs of Great Game Design

2D games, or games restricted to using two-dimensional graphics, have unique attributes that separate them from the manpower-intensive 3D games on the market today. Before you start making your very own 2D games, it is important to understand where they came from, what their key differences are, the most popular 2D game genres published now, and what makes for a fun and addictive 2D game.

Where Do 2D Games Come From?

The first games played electronically were *text-based games*. In text-based games, players read descriptions of scenes and characters and selected where they wanted to go and what they wanted to do there from a list of choices, much like *Choose Your Own Adventure* or *Fighting Fantasy* books.

Players of text-based games had to use their imagination to build the scene in their mind's eye and the selection process severely limited player choice. This did not frustrate too many players at the time, because no one foresaw the "sandbox" style of gameplay ushered in by game technology advancements in a couple decades. Most players of these early text-based games were thrilled to click together command phrases such as "use key on door." They would have been over-awed at the level of complexity and choice available in the *Sims* or *Grand Theft Auto* games!

When it came to games evolving from pure text to having visual graphics, the first graphics used were 2D *bitmaps*. Bitmap images are fixed-resolution images, generally scanned paintings or drawings, composed of tiny squares of color information called *pixels*. Each pixel has a dot of red, green, and blue information in it that sets the color tone. Bitmaps can give the illusion of continuous graduated color. However, as noted, bitmaps are fixed-resolution. This means that if you magnify a bitmap image, its quality will suffer and the edges will become blurry. That's why you blow up an image of the original Mario character from *Super Mario Bros.* and he looks like he was made out of bricks himself!

After years of technical innovation, 2D animation became more widely available and the graphics in 2D games improved, but in those first days, brightly lit squares of color on the screen composed the majority of most electronic games. It took years for the level of detail in arcade games to advance.

Sprites are typically animated 2D graphics. They are pixelated images of more than one frame of animation that cycles visibility of each frame algorithmically and can be anything from a player character's walk cycle to a fight sequence between two or more characters on the screen. Most 2D games that have animations are composed of sprites, which before 3D polygonal graphics improved, were the standard for electronic games (see Figure 2.1).

Electronic Game Firsts

The first computers were huge in size, very limited in processing power, had no real-time graphics, and very few people had access to them. Japan had simple

Figure 2.1
A classic example of a sprite-based game.

electronic games in arcade format before the U.S. In 1954 David Rosen, a Korean War vet, founded Service Games so that he could export coin-operated games, also known as *coin-op games*, from Japan to the United States. He later created his own games under the SEGA (SErvice GAmes) logo, which eventually stuck as the name for his company.

In 1961, Digital Equipment Corporation donated their latest computer to the Massachusetts Institute of Technology (MIT). It was called the Programmed Data Processor-1, or PDP-1. Compared to most computers at the time, the PDP-1 was comparatively modest in size, only as big as a large automobile!

Like most universities, MIT had several campus organizations, one of which was the Tech Model Railroad Club, or TMRC. TMRC appealed to students who liked to build things and see how they worked. They programmed for the PDP-1 for fun.

Figure 2.2
An image of Steve Russell's game *Spacewar!*

Steve Russell, nicknamed "Slug," was a typical science-fiction-loving nerd who joined TMRC. He put nearly six months and two hundred hours into completing an interactive game where two players controlled rocket ships. Using toggle switches built into the PDP-1, players controlled the speed and direction of their ships and fired torpedoes at each other. Russell called his game *Spacewar!* (see Figure 2.2). Thanks to "Slug," this game launched a whole new trend among programmers and became the predecessor for future arcade games.

Computer Space was the first commercially available arcade game and it was released to the public in 1971. It was a cabinet-style machine designed by Nolan Bushnel and, though it had many technological innovations, the gameplay itself was confusing and the product didn't become a commercial success. Using the profits made from the game, Nolan Bushnell left Nutting Associates and formed Atari.

Pong was the first successful arcade video game made. It was designed by Nolan Bushnell and Alan Alcorn and released in 1972. The gameplay is extremely simple: two players can play, each controlling a single vertical bar which can bounce back a moving dot much like a dummy's version of table tennis. Nolan placed the first *Pong* game machine in a local gas station. When he came back the machine had ceased to operate because it was so full of money. *Pong* became an instant success and it practically created the arcade video game industry all by itself!

Space Invaders, released in 1978 by Bally/Midway, created an overnight sensation. *Space Invaders* (as seen in Figure 2.3) was the first truly blockbuster video game. It brought the video games out of arcades and bars and into restaurants and

Figure 2.3
Space Invaders game cabinet.

corner stores, where video games quickly became a part of public consciousness and family lifestyles.

Note

The coin-operated arcade machines wouldn't record a higher score than 3,333,360 on the game *Pac-Man*. The first person to tally such a "perfect" score was Billy Mitchell of Florida in 1999.

Donkey Kong (as seen in Figure 2.4) was designed by Shigeru Miyamoto, Nintendo legend, in 1981 and was the first game I ever played. I found it in a coin-op laundry down the street from my house when I was just a kid and spent roll after roll of quarters there. *Donkey Kong* used the same hardware as an older video game called *Radarscope*. The idea of the game was to control a character called Jumpman, who tried to rescue a girl from a giant ape. Later on Jumpman was renamed Mario and starred in his own game called *Super Mario Bros.*, becoming the most famous and successful game character ever invented!

Out the same year, *Centipede* was designed by Ed Logg and Dona Bailey. It was the first arcade game to be co-designed by a woman! It is thought that its colorful

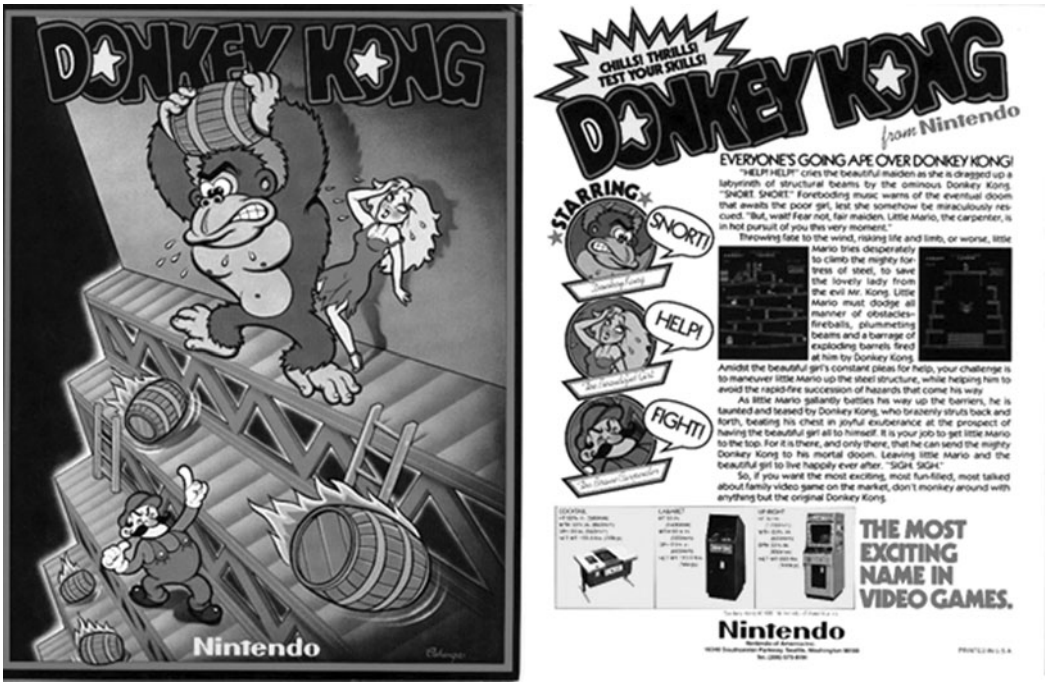


Figure 2.4
Promotional flyer for the *Donkey Kong* game.

graphics and fun gameplay made *Centipede* the first video game to be more popular with women than with men.

In 1982, Sega's *Zaxxon* introduced a 3D-lookalike view, making its game environment seem more realistic. This view is called *isometric perspective* in video games. *Zaxxon* had brilliant graphics for its time and became a big hit.

Starcom's *Dragon's Lair* (see Figure 2.5) was created by Rick Dyer and animated by cartoon animation professional Don Bluth in 1983 and was the first interactive animated film and first video game to utilize *laserdisc technology*. Its 2D graphics were way better than any of the games of its time, of Hollywood movie quality, and it had stereo sound. However, players complained of lack of choices in the game.

Its incredible graphics created huge media hype and in the following year Magicom/Cinematronics released another laserdisc animation-movie-game, called *Space Ace*, which was designed by the same team. Laserdisc players were very expensive and unreliable at that time and laserdisc games started to fade into obscurity in the middle of 1984, replaced by cartridge systems.

Figure 2.5
Starcom's *Dragon's Lair* game proved 2D animation could be done a whole lot better in video games.

About a decade later games reappeared in CD-ROM and DVD format, each similar to early laserdisc, and discs are now the most popular format for game distribution.

2D in a 3D World

3D graphics are virtual realistic scenes created from 3D polygon primitives rather than 2D vector or bitmap images. 3D graphics really took off big in the console market, where system resources were housed in a smaller, more controlled unit.

Wolfenstein 3D (1992) and *Doom* (1993), both from id Software, were designed with 2D components but programmed into a 3D environment. Each of the walls and characters in those games were similar to cardboard cutouts: 2D figures were pasted and made to stand up in a 3D environment. It would take longer before programmers started importing and using 3D polygonal models for real.

One of the first games to give indication of the switchover to 3D was Nintendo's *Donkey Kong Country*, released in 1994, which used 3D polygonal characters animated and pre-rendered to 2D images (see Figure 2.6). Critics gave the game much kudos for improving the look of the game and making the characters seem more ''real.''

Figure 2.6
Early examples of 3D pre-rendering in games felt a little flat but impressed a lot of audiences.

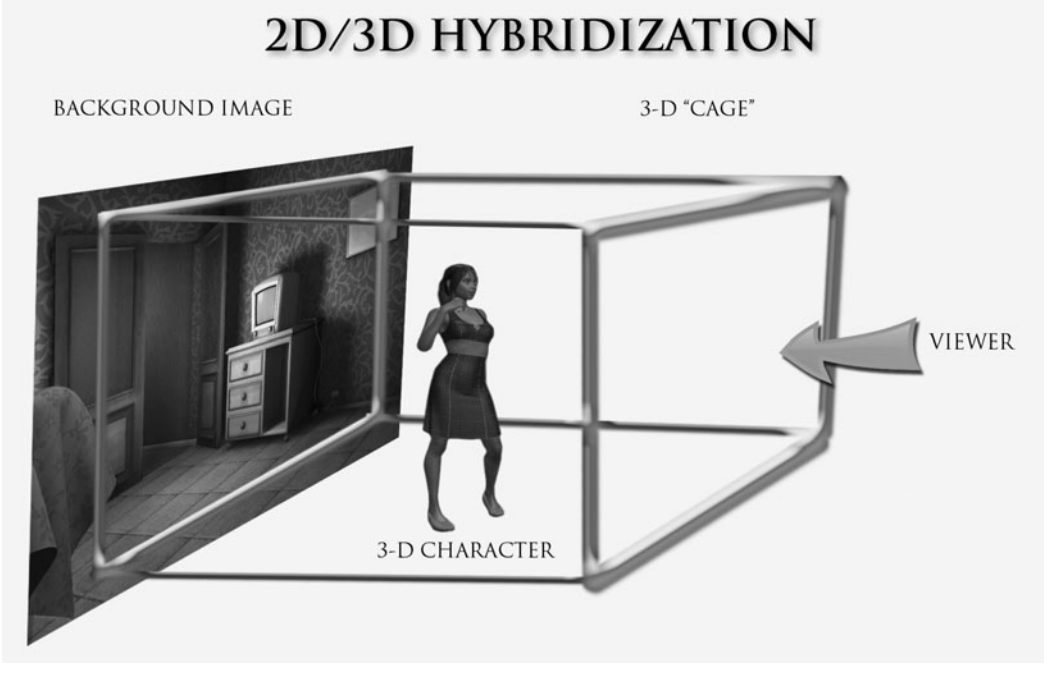


Figure 2.7
2D and 3D can work together to simulate environments.

Many of the earliest 3D polygonal graphics used a *2D/3D hybrid*. The first and second *Resident Evil* games, based on Shinji Mikami's idea and developed by Capcom in 1996 and 1998, were two such 2D/3D crossbreeds. The scenes were painted 2D backdrops with an invisible 3D box placed over the top of them so that players couldn't wander out of the scene's confines. The player characters and enemies were composed of 3D polygonal models. Look at Figure 2.7 for an example.

Just a few arcade games at the time attempted true 3D graphics. The first game to get global recognition was Sega's *Virtua Fighter* (1993), which brought fast 3D graphics to the world and changed the fighting game industry. However, most of the 3D terrain and characters remained very plain, without the texture resolution and bump or mip-mapping that would come much later.

As systems progressed, confines were removed that had once held game artists back. Nowadays practically all console games have 3D graphics, with hardly any exceptions, and the technology enhances as console machines become even bigger and badder, as seen in Figure 2.8.

This is all well and great, but you might wonder, "Where have all the 2D games gone?" Actually they've never left. 2D games are considered the most entertaining games on the market because of their staying power and focus. Especially

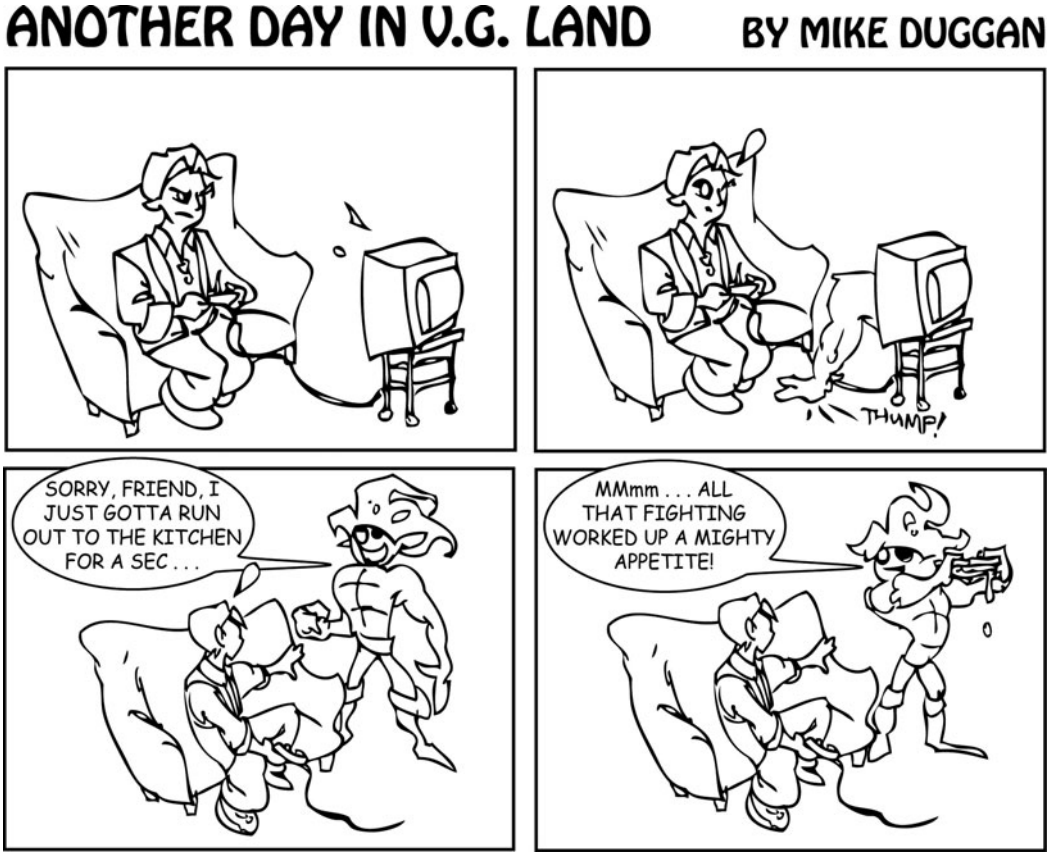


Figure 2.8
Astounding 3D graphics in modern games can knock out players.

where system resources are taken into account, such as web browser, mobile, and handheld games, 2D games are ripe for ease and versatility.

Game developers make full use of 2D game art and will continue to do so for the foreseeable future.

Popular 2D Game Genres

Retail outlets display games for their customers in alphabetical stacking order or grouped by publishers. But these methods don't give all the information possible to consumers. Some gamers may only want to play military shooter games, while others prefer sports games. There are many game types that have become traditional genres.

Just as literary fiction has its many genres (such as westerns, science-fiction, fantasy, and horror), games have their own unique blend of familiarity. Though the names and descriptions of game genres change frequently and even appear opposite on some gaming sites, most of the industry agrees on the following popular 2D game genres: action games, adventure games, role-playing games, strategy games, and more.

Action Games

Action games are games where the player's reflexes and hand-eye coordination make a difference in whether she wins or loses. The most popular action games include:

- **Shooters**—are games in which the characters are equipped with firearms and focus on fast-paced movement, shooting targets, and blowing up nearly everything in sight.
- **Platform Games**—allow the player's character to explore the upper reaches of a playfield by jumping on moving platforms and climbing ropes and ladders, all the while avoiding or knocking away enemies in a fast-paced animated world where one wrong step could spell disaster.
- **Racing Games**—feature fast vehicles along twisting tracks or difficult terrain in a race to the finish line. Some have mayhem and combat, with the vehicle being a weapon in and of itself.

- **Sports Games**—feature rules and team meets just like the real-world counterpart sports.
- **Fighting Games**—focus on competing against opponents in arena combat. The controls are often limited to a number of combination moves.
- **Stealth Games**—are games that reward players for sneaking into and out of places without being seen and striking enemies silently. Though these games do have combat in them, the major focus is on avoiding direct confrontation.

Adventure Games

Adventure games traditionally combine puzzle-solving with storytelling. What pulls the game together is an extended, often twisting narrative, calling for the player to visit different locations and encounter many different characters. Often the player's path is blocked and she must gather and manipulate certain items to solve some puzzle and unblock the path.

Zork was one of the first interactive fiction games ever played on a computer. The name *Zork* is hacker jargon for an unfinished program, but by the time Infocom's *Zork* (see Figure 2.9) was released and going to be named *Dungeon* in 1979, the nickname *Zork* had already stuck. For many, the name *Zork* conjures up dim images of a computer game prehistory, before graphical adventures had become the norm. *Zork* was also one of the greatest adventure games of all time and set several precedents for the genre.

Adventure games primarily center on story, exploration, and mental challenges. Most, if not all, adventure games don't even have violence in them. Many have players solve mysteries through gathering up specific clues, as in Toshimitsu Takagi's 2004 game *Crimson Room*.

Role-Playing Games

Role-playing games (RPGs) got their start in pencil-and-paper in the 1970s with the late great Gary Gygax' *Dungeons and Dragons*.

I was introduced to tabletop RPGs in my teens, and they got me interested in computer games as the technology for them improved. Today's more complex computer role-playing games like *Neverwinter Nights*, *World of Warcraft*,

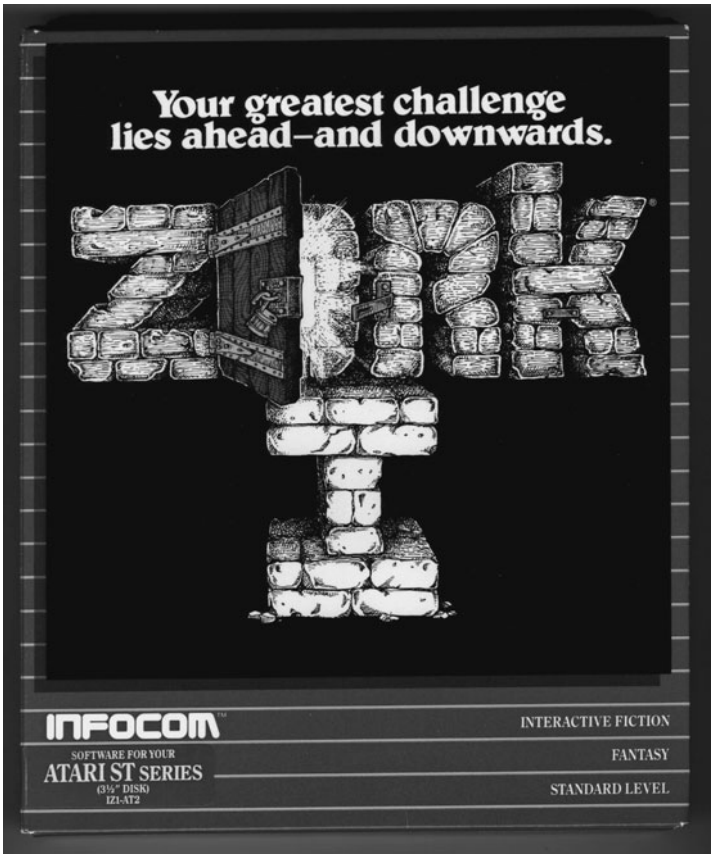


Figure 2.9
Infocom's *Zork I* game.

EverQuest, and *Elder Scrolls IV: Oblivion* offer players pretend worlds with an amazing level of immersion. A player like Sally creates and enhances her own character from templates or from scratch. This character (see Figure 2.10) usually has a number of different game statistics used to resolve game actions, including statistics for health (expressed as hit points), physical attack strength, physical defense ability, and many more (such as intelligence, charisma, dexterity, and so on). A point system for experience is used to raise up these statistics over time, making the character stronger and better at surviving overall.

Most RPGs cover a core story arc that gives Sally a goal to achieve, but others are open-ended and allow Sally to go where she pleases and accept or deny quests offered her. The main goal of most RPGs is for Sally to gain enough experience or treasure for completing missions and beating monsters to make her character stronger. Along the way she'll travel to many locations and meet

Figure 2.10
A classic example of RPG adventuring often includes character creation.

many *non-player characters* (NPCs), which are characters not controlled by the player but by an artificial intelligence programmed into the game itself. NPCs can both hinder and aid the player as the game developer sees fit. A lot of RPGs focus attention on the player conversing with NPCs, often through multiple-choice dialog trees. NPCs generally send the player on quests, trade the player special items, or impart information the player needs. Some NPCs can become enemies.

Strategy Games

Strategy games envelop a great deal of mental-challenge-based games, where the player builds an empire, fortress, realm, world, or other construct, manages the resources therein, and prepares against inevitable problems like decay, hardship, economic depravity, revolution, or foreign invaders. Many strategy games, like Microsoft's *Age of Empires* game released in 1997, are *isometric perspective* games. This means that even though they are 2D they give the illusion of being 3D. Several of these games will have a military unit component and feature a dark screen that is opened up as the player travels the terrain, called the *fog of war* (which offers uncertainty as to the enemy's location and actions).

Figure 2.11
Clever strategy games can teach players to become capitalists.

A newer 2D strategy game, pioneered by Gamelab's *Diner Dash* in 2004, features a customer-service core play, where the player is in charge of an eatery and must please customers in order to make the biggest profits. This style of strategy game teaches players money sense, management, and prioritization skills (see Figure 2.11).

Other 2D Game Genres

Besides the 2D game genres already mentioned, there are many more:

- **Advertainment Games**—advertise a particular brand or service and are generally developed as part of a public relations' campaign.
- **Artificial Life Games**—make players care for a creature or virtual pet. Nintendo's *Nintendogs* for the Nintendo DS is one of the best known artificial life games on the market.
- **Casual Games**—include such traditional games as Chess, Poker, Texas Hold'em, Solitaire, mah-jongg, and trivia.
- **Puzzle Games**—never have much of a story but instead focus on mental challenges. Popular puzzle games include *Bejeweled* and *Tetris.*
- **Serious Games/Edutainment Games**—facilitate schools by teaching subjects in the guise of having fun, or they can help companies instruct their employees.

Licensed Games

When movie studios realized that games could be a profitable way to reach their market, they started charging hefty licensing fees for their properties.

Unfortunately, game developers sometimes spend so much time and money acquiring a licensed property (LP) to build a game based on a blockbuster film they don't have enough budget leftover to make as good a game as they might like. There's also a minute window of opportunity to capitalize on the movie's publicity to exploit the game's release potential.

A keen example of a poorly done game intended to follow on a movie's coattails has to be *E.T. the Extra-Terrestrial* released by Atari Inc. for the Atari 2600 game system in 1982. The game has been viewed both as one of the worst games ever made and the biggest commercial disappointments in game industry's history. As a result of overproduction and profit-making failure, hundreds of thousands of game cartridges were purportedly buried in a New Mexican landfill.

What Makes a 2D Game Great?

2D games are great because they don't require a lot of manpower to create 3D models and backgrounds and render complex 3D scenes; you don't have to purchase a 3D editor such as Autodesk's 3ds Max or Maya (each of which cost well over a thousand dollars for a single user license!); plus, 2D is a designer's choice when the game must be fast, fun, and relatively smaller in file size.

However, when speaking of greatness in relation to 2D games, there are many other elements to consider. Some games come out, get downloaded and played just a few times, and then fade into obscurity. Worse, there are other games that get nit-picked to death by critics before they are even released and negative press drives them into submission. Then there are those games that rise out of total obscurity and become household names and everybody's playing and talking about them.

Let's look at what special ingredients go into making a 2D game rise above the rest.

Interactivity

Whereas a movie might satisfy a consumer in its two hours of playtime, games are expected to offer much more entertainment for a lot longer time. What sets a game apart from reading books or watching movies? Interactivity! Have you ever played an obnoxious game that appeared to be one long cinematic scene after another with short pauses in-between where you got to explore the environment as your character before hitting just another cinematic scene? These games

behave more like abstract film projects than fun games, and that's because games are meant to be interactive!

Players don't want to be told a story; they want to tell the story themselves. Listening to long-winded exposition, forced to watch long animated sequences, and even talking with characters should *always* be secondary to exploration, combat, manipulation, and problem solving. In other words, story is supplementary to interactivity.

Empowering the Player

Let's call our game player Sally. When Sally picks up her game controller or takes over her keyboard and mouse, she wants to be able to explore make-believe worlds, encounter responsive creatures, and interact with her game environment in ways she can't get out of watching a show or reading printed words. If you fail to empower Sally with interactive control, you fail as a game designer.

Part of empowerment is giving the player choices. Sally should not have all her choices made for her arbitrarily. The choices Sally is given should be reasonable ones. Don't ask her to go in the door marked "Great Stuff Inside" and then have a brick wall on the other side of it. Likewise, don't ask Sally to choose between getting a magnificent sword and a pile of junk, because she'll pick the sword every time.

The environment in 2D games must be somewhat reactive. Having *reactive environments* means that the game world responds to the player in logical and meaningful ways that help immerse the player in that game world. For instance, if the player sees a neat looking door and wants to open it, he should be able to or the game should let him know, "This door is locked. To open, you must find the key." This empowers the player to explore the game's environment and to treat it as if it were its own real self-contained world.

Immersion

Have you ever played a game that you focused on so hard that when your friend called or your parents interrupted your concentration, you realized with shock you'd been playing for hours straight? Have you ever been playing a game so intently you didn't want to stop? If the answer to either of these was "Yes" then it's because you've discovered another key element of popular games: *immersion*. Immersion creates addictive game play by submerging players in the

entertainment form. With immersion, you get so engrossed in a game that you forget it's a game! You lose track of the outside world. You also believe the intrinsic game rules more stringently than if you were not so immersed, a trick called the *suspension of disbelief*.

The following are the different depths of immersion the gamer (again, we'll use Sally) feels when playing a game:

- **Curiosity**—Sally feels a slight but fleeting interest in the game.
- **Sympathy**—Sally is paying attention to the game but is still not personally motivated; she can put the game down if she wants.
- **Identification**—Sally identifies with her character and suddenly has an invested interest in the game's outcome.
- **Empathy**—Even though the characters are make-believe, Sally shares a strong emotional connection with one or more of them and wants to see the best ending to the game.
- **Transportation**—This is the "plenary state" or dream-like trance that you enter into whenever you are really intent in playing a game. The game becomes more real to Sally than the room where she's playing it.

Emotioneering

You will learn that making players care about what happens in a game is not always an easy task. Fashion design guru Marc Eckō broke onto the video game world in 2006 with *Getting Up: Contents Under Pressure*, a game about a graffiti artist. Eckō called games "emotional entertainment products" because he considered games to be a form of entertainment unique in that it's the only form of entertainment that forces players to interact with it on a closely personal and emotional level (as seen in Figure 2.12). Emotions can be used to make players care about the games they play.

Eckō is not the only game creator out there who shares this viewpoint. Screenwriter David Freeman started the Freeman Group, which studies the many ways writers can put emotions into games. Freeman pioneered *emotioneering*, a cluster of techniques seeking to evoke in gamers a breadth and depth of rich emotions. These emotions not only create stronger immersion, but they also generate control points for the designer to maneuver the player through the game.

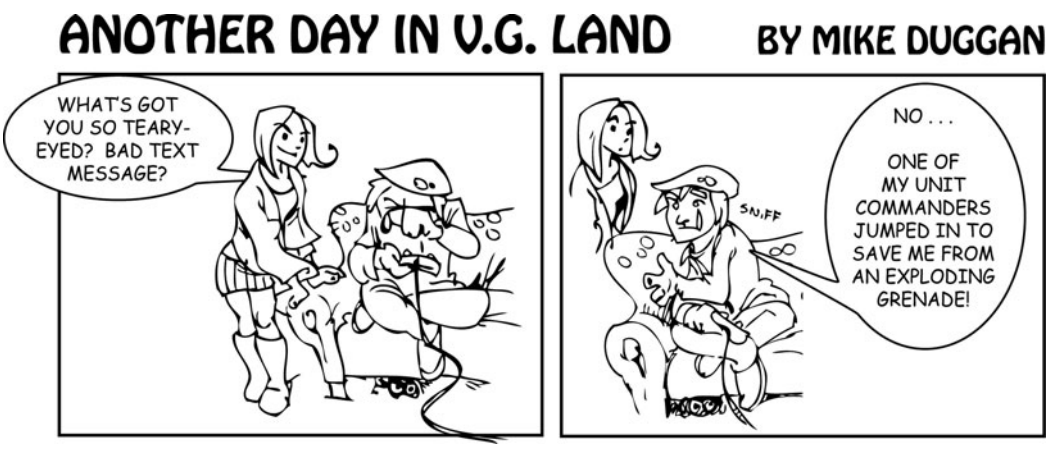


Figure 2.12
Games can use a player's emotions to get their attention.

Most of the emotioneering tricks are subtle triggers you can plant in your game's story, including:

- Keeping the plot twists coming. Remember: ''Out of the frying pan, into the fire.''
- Having the other characters recognize or refer to one another as if they were real people.
- Giving the player ambivalent feelings toward an ally or enemy character, like loving and hating them at the same time.
- Forcing the player to do something evil or otherwise violate their character's integrity.
- Having the player discover she's been tricked or betrayed by an ally.
- Setting up incongruous events (like when the main character of Nintendo's *Chrono Cross* suddenly switches places with the main villain and has to gain new allies after losing all his friends).

Game Challenges

A real game wouldn't be a game if it didn't offer the player a challenge. The types of challenges games offer vary widely, from the accumulation of resources to puzzles to self-preservation. Many challenges are staples of the game genres they belong in; for instance, ''fetch quests'' are challenges popular in adventure and

role-playing games, where the player is asked to go and fetch some object in order to complete a task.

The most important thing you can remember about challenges is that they are met with bravado. All the cheats that you see online for games have been discovered and used by gamers who have learned the one almighty truth when it comes to winning video games: ''Don't play the game, play against the game's underlying programming!''

Dennis Wixon who worked with Microsoft Games Studios in Redmond, Washington, says about challenges, ''You're always trying to get the right level of challenge. You can't be too simple or it's not fun. [Nolan] Bushnell's famous quote is something along the lines of, 'A game should be easy to pick up and impossible to master.' We want that sweet spot where there's always another threshold to cross. In *Halo 1*, as we improved targeting, we found it was too intelligent and too simple. It was pretty straightforward for the Bungie team to fix that. . .''

Andrew Glassner calls this process the *game loop*, a cycle of repetitive steps the player takes to win at any given game challenge:

1. Player observes the situation
2. Player sets goals to overcome the challenge
3. Player researches or prepares
4. Player commits to plan and executes decisions
5. Player stops and compares the results of his actions to his original intention
6. Player evaluates the results
7. Player returns to step 1

Quests

Quests are special sets of challenges that take place in both stories and games, thus linking narrative and play. Quest games, including the *King's Quest* series, have quests that make up activities in which the players must overcome specific challenges in order to reach a goal, and when players successfully surmount the challenges of the quest and achieve the quest's main goal, the player's actions bring about or unlock a series of events comprising the game story.

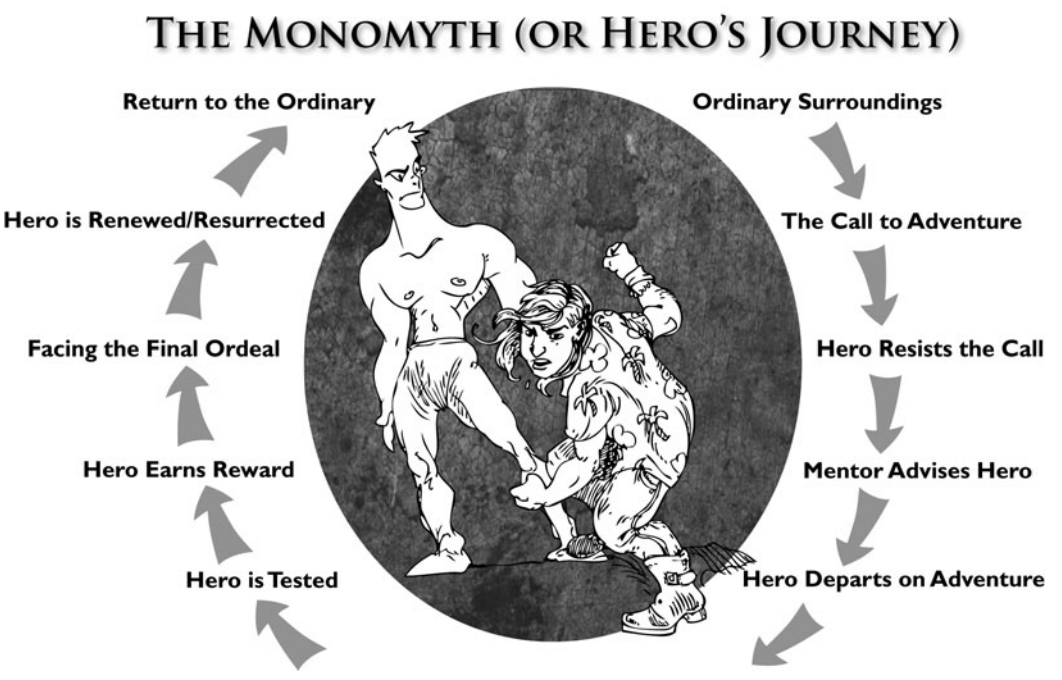


Figure 2.13
The predictable steps in the monomyth.

Game writers often cite Joseph Campbell's monomyth or "hero's journey" as a pattern for their quest games (see Figure 2.13). In the "hero's journey" there are several legendary steps, which you can imagine as a staircase, where the first step starts with the hero in his own world confronted with a terrible evil threat requiring him to go where he's never gone before or do things he's never done before. He must find allies, fight enemies, and learn something uniquely special about himself along the way, before eventually overcoming the terrible evil. This mythic story structure forms the basis for the majority of our ancient legends and our current Hollywood story compositions. For a look at this pattern, you should peer at *Star Wars* and *The Lord of the Rings*.

As Jesper Juul explained in *Half-Real: Video Games between Real Rules and Fictional Worlds*: "Quests in games can actually provide an interesting type of bridge between game rules and game fiction in that the games can contain predefined sequences of events that the player then has to actualize or affect." You don't have to write a quest game to use quests to improve your games. Many action games, including 2K Games' *BioShock*, have implemented quests to reveal narrative and create further depth of player experience.

We will look at creating game challenges and quests in more detail in Chapter 7, ''Adding Game Challenges.''

The 4 Fs of Great Game Design

Ultimately, it is not possible to design an ideal game that pleases everyone and makes you a load of money, because not everyone enjoys playing the same games. But never fear! There are 4 Fs of Great Game Design that will enhance your chances of pleasing the most gamers. The 4 Fs, which are listed in priority, should be considered whenever you have to make any design decision.

The 4 Fs of Great Game Design are *Fun*, *Fairness*, *Feedback*, and *Feasibility*.

Note

Keep in mind there's another F that is the sworn enemy of the game designer: *Frustration*. You do not want to *ever* foster frustration in your gamer. Games are meant to challenge the player, but the player must be able to overcome the challenges without surrendering to the gnawing feeling of frustration! Look out for game bottlenecks, problem areas, and impossibly difficult challenges.

Fun

Tip

"We could have lost an eye! Then no more fun and games. Except for pirates. Lot of pirates have just one eye. They have lots of fun and games."

—Hank Yarbo, *Corner Gas* (2004)

Our player Sally is willing to put in as much work as required if she gets back enough high-quality fun. Fun, after all, is what games are all about!

If you find that your game is not providing the player with high-quality fun, stop what you're doing right now and go back to the drawing board. Remember to build your game on the premise that every part of it must be fun. A trick to keep in mind is that if you are not having fun making the game and you don't want to play it the minute it's finished, Sally probably won't have fun playing it either—so enjoy the building process!

Fairness

Unfair challenges in a game can lead to frustration. Avoid frustration by making the game easier for our player (Sally). Don't remove challenges from the game completely, but relieve the build-up of tension that could potentially lose the player's attention.

Different people, and indeed different players, will have varying notions of what is and is not fair. No matter what meter is used to measure fairness, it is an unspoken social code when it comes to games.

Never set Sally up so that she has to perform a complicated set of maneuvers to get to the top of an Aztec temple, only at the last minute having her fall all the way back down to the ground, forcing her to start all over again. Endless repetition can be absolutely maddening, so don't let Sally fall into that rut. Players will sometimes change the rules of the game if they perceive that the rules are unfair or permitting unfair behavior in the game. This is partly why we see so many cheat codes.

One sign that your game is unfair to its players is if they have to resort to using cheat codes or complain to you that the obstacles in it are impossible to overcome. Rely on play-testers testing your game before calling it "done" and releasing it to the general public, or else it will come around to bite you later.

Tip

"Geez, Brian, this isn't what I was expecting, I thought being a hero would be all fun and games."

—Peter Griffin, *Family Guy* (1999)

Feedback

Feedback is one of the primary mechanisms of any human-computer interface, especially games. Providing Sally with adequate feedback will help Sally know what to expect out of your game and frames the choices she will make.

If you've ever heard psychologists talk about *positive/negative reinforcement*, then you already have a clue how to implement feedback. If you want Sally to enter the haunted house full of phantasms and defeat all the phantasms, you have to give Sally some motivation for doing so, and when she does clear the house you have to give her some significant reward. Likewise, if you don't want Sally to do something, like enter the Forbidden Courtyard, you have to set up reasonable punishments for her if she tries.

Some of this feedback can be purely psychological, while some can appear physically during the game. Using the Forbidden Courtyard example, you could show Sally that everybody who's tried to enter the Courtyard before has been speared right through by trap floors. Sally won't try it if she doesn't think she'll be able to survive. This is a psychological trigger. You could likewise use a physical

trigger, such as having the Forbidden Courtyard blocked off by crumbled pillars and masonry debris.

Adequate and timely feedback will improve the way your game is received.

Tip

> "This certificate says that I have the fortune now! And there's nothing you can do about it! [*aside to Violet*] What do you think? Too diabolical? Give me some feedback."
>
> —Count Olaf, *A Series of Unfortunate Events* (2004)

Feasibility

Something that is feasible is something that has the quality of being doable. This means your game must make sense or it will fail.

At the same time, most games don't make a whole lot of sense; take Super Mario Brothers for example: you play an overweight plumber who kills strolling mushrooms and kamikaze turtles by jumping on them! The game doesn't try to make a lot of sense, but it *is* fun and it *is* also consistently feasible. What I mean is that the game rules never change. Mario can gain bonuses from grabbing coins, and Mario must avoid the fireballs because they hurt him. The same would not be true if halfway through the game, without preamble, Mario could be hurt by touching the coins and could gain bonuses from hitting fireballs!

Keep your game rules consistently feasible or you'll lose your player to frustration.

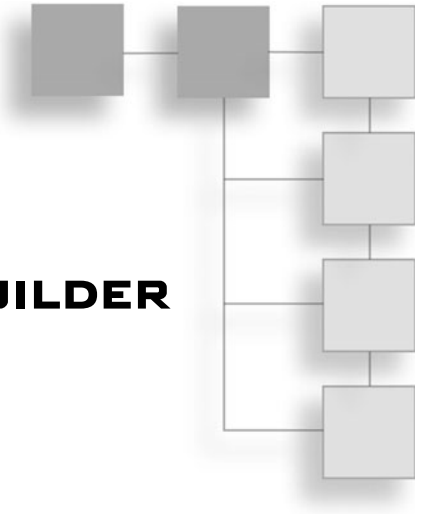
Review

After reading this chapter, you should understand the following:

- Where 2D games came from
- Some of the most historic electronic game firsts
- What the difference is between 2D and 3D game graphics
- What the most popular 2D game genres are
- How empowering the player, immersion, and emotioneering create interactivity
- How game challenges should function
- What the 4 Fs of Great Game Design are and why they are important

This page intentionally left blank

CHAPTER 3



The Torque Game Builder

In this chapter, you will learn:

- What the Torque Game Builder program is and who started it
- The major components of TGB, including the 2D Toolset and more
- Other products from GarageGames
- Where to find further resources

GarageGames is a software company and Internet publishing label based in Eugene, Oregon, committed to making applications for game designers to make games of their very own. GarageGames' name is purposefully based on the term "garage band" to target the independent game developer market (i.e., guys and gals making electronic games in their spare time from the garage or basement). The number one goal of GarageGames is to offer licensing of game engines and publishing to these indie developers, which includes you.

The founders of GarageGames, including Jeff Tunnel, Rick Overman, and Tim Gift, previously ran a game development company called Dynamix that was started in 1984 and made games such as *Earthsiege*, *Starsiege*, *Tribes*, and *Tribes 2*. The Torque Game Engine, or TGE, is a modified version of the 3D game engine originally developed by Dynamix for their 2001 first-person shooter game *Tribes 2*. Their triple-A title 3D game engine is one of the most comprehensive and affordable on the market today and a license to use it is available from GarageGames.

The Torque Game Builder

Although the Torque Game Engine is excellent for making 3D polygonal graphic video games, we want to make 2D games. GarageGames' *Torque Game Builder* is a separate game engine that has been designed specifically for making 2D sprite-based games.

Say you have a concept for a game, but not enough time, knowledge, or assets to start from scratch and create your own tools and technology. Torque Game Builder offers you many features, including animated sprites, flexible tiles, a particle system, swept-polygon collision, rigid body physics, and hardware-accelerated 2D rendering. TGB is the perfect entry into game development. As its slogan goes, "Make it fast, make it fun!"

The Torque Game Builder has been used successfully to develop several commercially published games (see Figure 3.1), including but certainly not limited to the following:

- *Atomize*
- *Fortune Tiles*
- *Gold Fever*
- *Kachinko*
- *King Kong: Skull Island Adventure*
- *LEGO: Bricktopia*
- *Phantasia*
- *Puzzle Poker*
- *Rack 'em Up Road Trip*
- *Trick Ball*

Figure 3.1
Games made with the Torque Game Builder.

Licensing

Game Builder is not free, although a free-trial demo of the software is available for your use. The licenses depend on how intensely you want to get into creating your own 2D games. In order of power, from entry-level to professional, the Game Builder licenses are as follows:

- The Indie License
- The Commercial License
- The Pro Licenses

As of Torque Game Builder version 1.7.0, the *Indie License* allows the engine to be used by independent game developers for $100 per programmer, provided that said programmer is not employed by a development company with annual revenues of $250,000 or more. This licensing model is intended for low-budget designers as it saves them the time and effort of programming their own game engine without requiring a large amount of money to purchase the license. The Indie License is binary-only and does not include source code. It also requires that you display the GarageGames logo before the game starts up in all of the games you release. As an amateur game developer, you probably won't need to go beyond the Indie License, so it's your best bet.

With an Indie License, you cannot transfer license ownership, though this limitation is omitted if you buy the *Commercial License*, which costs $495. The Commercial License also offers you access to developer training through Torque Boot Camps.

Note

Torque Boot Camps provide you with three intense days of training in the GarageGames software applications, including the Torque Game Builder. You will come out knowing more than you ever thought possible about the underlying concepts, skills, and tricks-of-the-trade used with Torque. Torque Boot Camps, however, are only available to those who purchase the Commercial License of the Torque Game Builder or Torque Game Engine.

The *Pro Licenses* are not binary-only; they contain full source code for the game engine. The Indie Pro License costs $250 and the Commercial Pro License costs $1,250.

No matter which license you decide to go with, you pay no royalties ever with GarageGames' royalty-free licenses. You can publish your game anywhere you

want, whether it is on the Internet or in stores. It's your game, so you make the decisions.

Torque's Features at a Glance

Your first impression might be that the Torque Game Builder appears to target the making of side-scrolling action games, but this belies the true power of Game Builder. Sure, you can make a side-scrolling action game—you could even remake *Super Mario Bros.* if you wanted to—but you can *also* create any game of any genre in 2D that you want, and if you so desired and had the time to put into it, you could use Game Builder to create a professional or educational product that has nothing at all to do with games. There are several developers right now building educational or corporate training tools with Torque Game Builder.

Note

Sidescrolling means that a game fits the retro feel, where the player's character starts on the left-hand side of the screen (usually) and the player navigates the character to the right-hand side of the screen (see Figure 3.2). The invisible camera that the game is viewed from is locked onto the player character, following its movements. Sidescrolling games usually fit snug in the action category of games, specifically the platformers, where running and jumping, avoiding obstacles and bad guys, and timing jumps onto moving platforms are an integral ingredient.

Figure 3.2
A classic sidescrolling game.

Built-In Components of TGB

All the Torque Game Engines share a unique suite of tools in their toolkits that empower you to build the very best indie games you can. These include:

- **2D Toolset**—easy-to-use 2D game editors, including a Particle Builder, Level Builder, GUI Editor, and Tile Editor.
- **Other Built-in Features**—such as a Torque scripting language, collision detection system, physics engine, and integrated online multiplayer support.

Note

The Game Builder is a *software development kit*, or SDK, and it is therefore full of lines of code after lines of code, libraries, demos, kits, resources, and everything you need to make your own games. However, be forewarned, because—as with many SDKs—it takes some time to learn. Yet that's exactly what makes Game Builder the powerhouse that it is, and when all is said and done, you can manage it and put yourself in the game creator's chair!

2D Toolset

You can create stunning visual effects with the Torque Game Builder 2D Toolset. You can set up particle effects with the Particle Builder, maneuver tiles with the Tile Builder, and use the intuitive drag-and-drop interface of the Level Builder to create complex game levels. TGB's 2D Toolset makes it fast and easy to generate a lot of content with minimal scripting.

Particle Builder When you think of particles used in games, think of fireballs, smoke, fog, ashes, snow, rain, and other minor details that a game would not be the same without (see Figure 3.3). There are three elements to any particle system, and they include:

- **Particles** themselves are very small and efficiently animated sprites that change properties over time.
- **Emitters** spew particles in a more convenient manner.
- **Effects** include a grouping of particle emitters, and control these emitters over time.

Effects and emitters can be defined as either static or time-dependent, and if they are time-dependent, they are controlled by a time-graph system. Furthermore, particle effects are a subclass of scene objects, meaning they can be moved, scaled, mounted, assigned to groups or layers, and even exhibit physics.

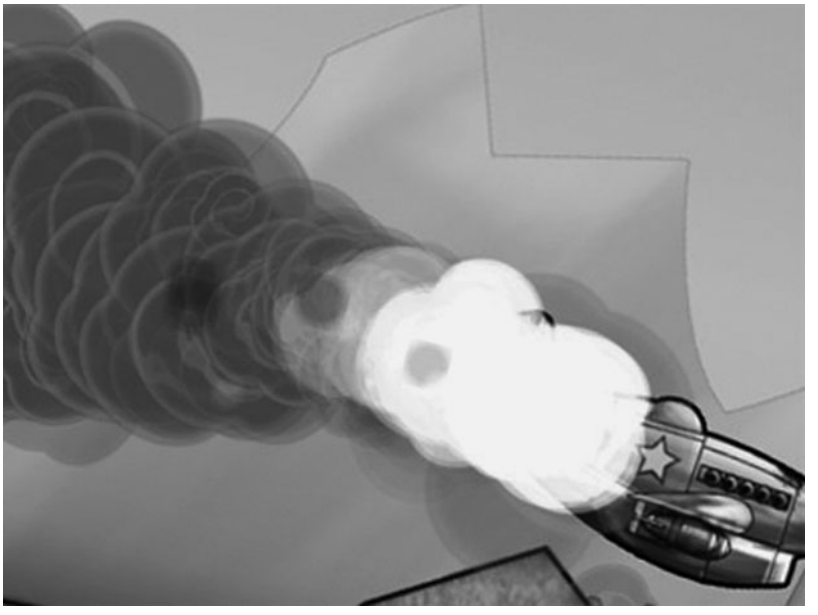


Figure 3.3
Particles can appear as any image map that is used repetitively in deciminating value, such as this smoke trail.

Level Builder Drag-and-drop objects and effects from a library of thumbnail previews, then edit those objects or effects directly inside the level editor (see Figure 3.4). Click on an object and view the properties of that object, including collision factors. Click on a particle effect and view the parameters for that effect. It's fast and easy!

GUI Editor A *graphical user interface*, or GUI, is the look of the shell extension of a game, including the windows, interactive menus, and heads-up display. Some games attempt to design complete transparent interfaces, where

Figure 3.4
The level editor of Torque's platform game maker.

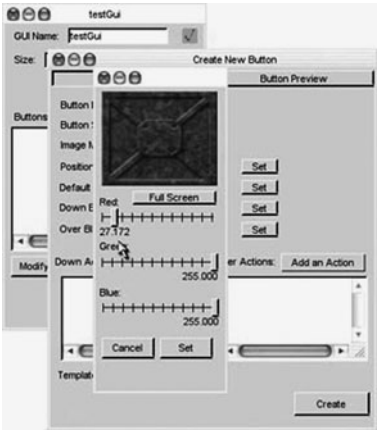


Figure 3.5
The GUI editor.

these windows and options are hidden or nonexistent, because this has been shown to increase immersion. But the majority of game designers create theme-inspired GUIs that go with their game and make them very artsy. The GUI editor (see Figure 3.5) makes it easy to create a custom GUI for your game and it features broad font support and allows you to script your own controls.

Tile Editor *Tiles* in games are flat 2D images, which can be static or animated, and they are repeated over and over in the background to make up the terrain or surface area. Objects can also collide with your tiles, and those collisions remain true to Newtonian physics responses. TGB supports tiles that can be set to "active." *Active tiles* are those tiles that dynamically affect the game world. This interface (see Figure 3.6) helps create realistic 2D levels.

Camera System Due to the influence of photography and cinematography, game designers refer to the way they control the visibility of a given scene in their game as a *camera*. Though this camera is often an invisible, arbitrary object in the scene, it defines the way the user sees the action on his screen. The camera system in TGB is extremely flexible:

- You can set view screens to cover a specific zone with a specific zoom level.
- You can give the view screen a target position and it can move there dynamically over time from its existing position.

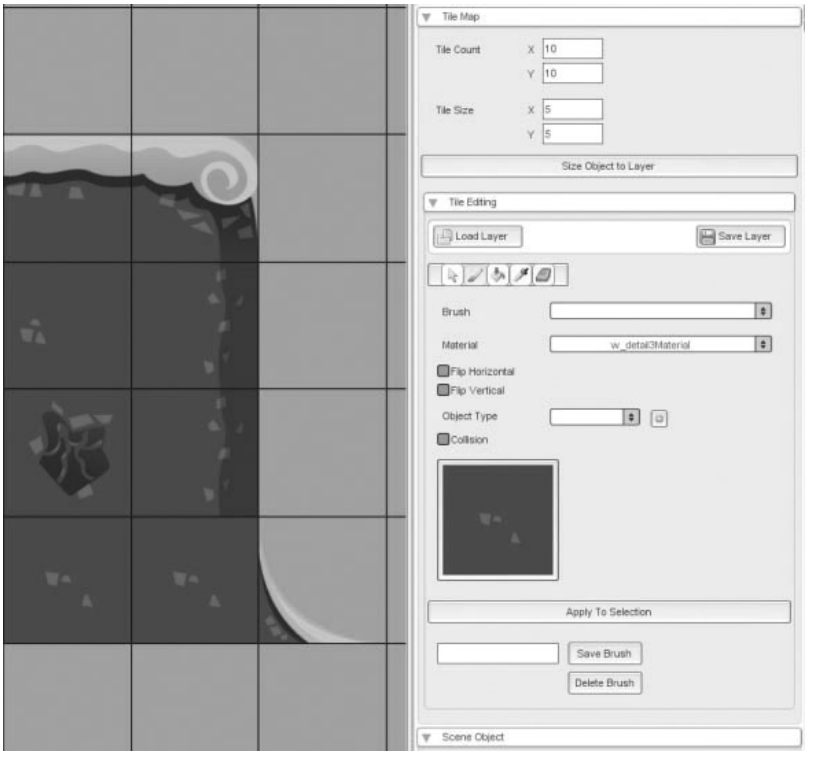


Figure 3.6
The tile editor.

- You can mount the view screen to any object, such as the player avatar. You can make the camera follow an object from a fixed distance, or react to forces on the object.
- Also, TGB allows for classic camera effects, such as shaking in response to an earthquake or bouncing up and down when the player's race car is traveling over rough terrain.

TGB's camera system is a powerful foundation for building amazing and artful games. The player's perspective is the basis for their interaction with the world, and controlling it controls how they view the game.

Sprite Support Sprites can be scaled, rotated, and divided into two major classes: those with animation capabilities, and those without.

Animated sprites depend on animation datablocks and animation controllers in order to play animations. *Animation datablocks* load frames from an image map,

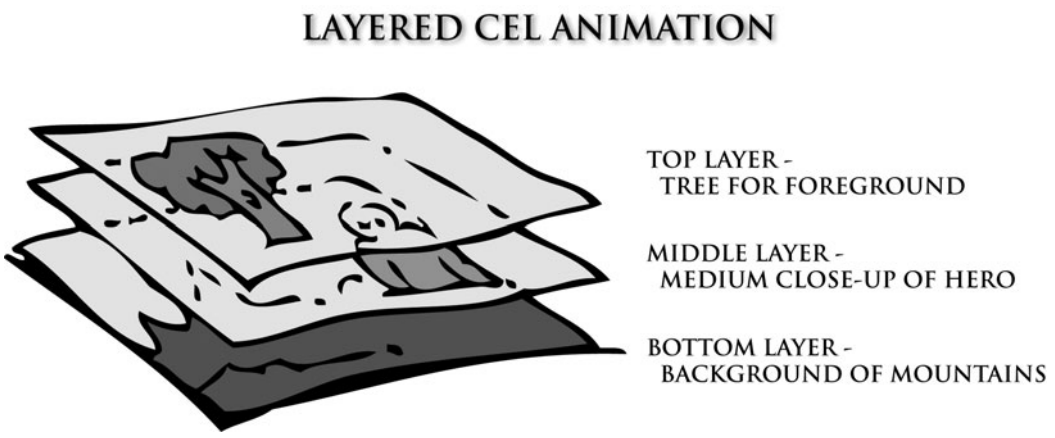


Figure 3.7
Layering objects in a scene can give you, the designer, more control over the look.

delineate an animation as a chain of frames, and define the play time of the animation. The Animation Controller in TGB controls the animations of sprites (as well as particles and tiles).

Static sprites are those without complex animation capabilities. They can still move about the scene, have full collision and Newtonian physics applied to them, but they don't have a powerful animation system.

Layers Each thing in the game world must be drawn. By changing the order in which things are drawn, different effects can be achieved.

Think of the layers of an onion. In traditional cartoon animation, characters are painted on one cel, or transparent sheet, and placed over another and another, with the background being the very bottom cel to be animated (see Figure 3.7). Having objects sorted to different layers helps to establish depth and makes sure that one character can't walk over the top of another one when he shouldn't be able to. Layers serve the same important task in game rendering.

Your TGB game objects start out assigned to layers, but the sort order for layers can be changed dynamically, and objects can be re-assigned to new layers at any time. Layers come in handy when setting up parallax scrolling.

Parallax Scrolling *Parallax scrolling* consists of dissimilar planes of graphics that scroll by at unlike rates of speed depending on their perceived relation to the viewer. Parallax scrolling creates an illusion of depth in an otherwise 2D environment. In TGB, this is done by defining more than one scrolling background and sorting them into different rendering layers at different scroll speeds.

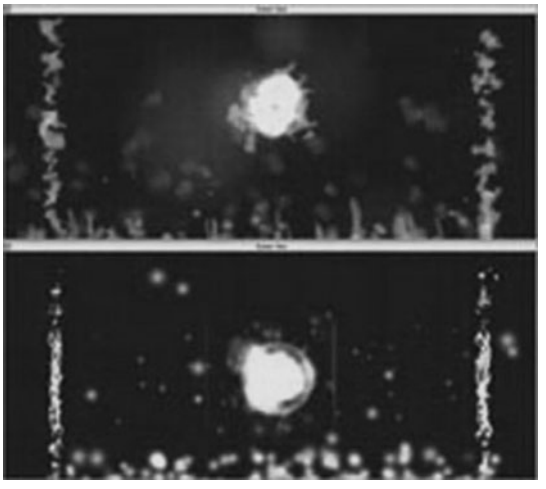


Figure 3.8
Viewing multiple scenes at once.

Scene Graph Torque Game Builder's *scene graph* is a data structure that keeps track of items in the scene. The hierarchical nature of the scene graph allows you to change game-wide assets, or only those pertaining to a specific class of objects. TGB culls the objects that won't be seen on screen; this culling of off-screen objects averts superfluous drain on CPU and RAM, keeping the graphics loading smoothly.

Multiple Scenes Run multiple scene graphs simultaneously and multiple windows drawing from the same scene graph (see Figure 3.8). This allows you to edit faster than the average game designer and also means that you could set your game up for multiple views.

Other Built-in Features

The Torque Game Builder SDK also comes preset with an array of built-in features that will make your game creation faster and easier, including Torque's very own scripting language, TorqueScript, an integrated collision detection and physics engine, TorqueNet Lite (allowing for online multiplayer games), CodeOnce (meaning that when you create a game in TGB it can be played on multiple operating systems), and a huge growing community of developers just like you!

TorqueScript *TorqueScript* is a simple and easy-to-learn C++ like scripting language. TorqueScript sets up the framework that defines how all the elements of your game should behave. Your game can be programmed in TorqueScript,

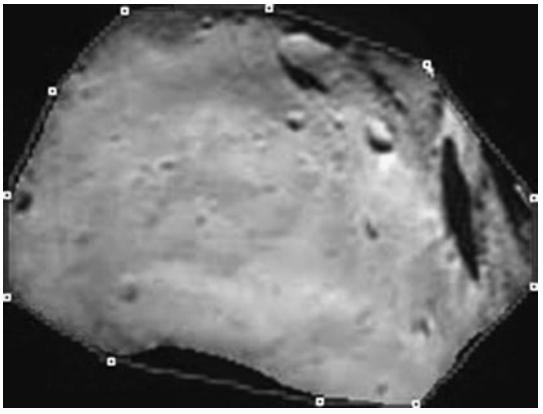


Figure 3.9
A tight-fit polygon sets up collision for this asteroid.

although you can use C++ for additional snippets. The third-party supplemental tool Torsion, developed by Sickhead Games, permits you to program in TorqueScript using fast syntax editing. You can read more information about it online at http://www.sickheadgames.com/torsion.php.

Collision Detection When one object collides with another, it doesn't automatically pass right through the other object like walking through walls. There's always a causal reaction when objects hit one another. In other words, your warlock player character will bounce off a wall when he comes in contact with it instead of walking straight through it.

With TGB, you create tight-fitting collision polygons and set up collision causation for a more realistic environment (see Figure 3.9).

Through the TorqueScript, objects can generate some sort of reaction upon collision. The most obvious and oft-used reaction would be to generate flashes or explosions at the point of contact between two objects or to make one object hurt another when it comes into contact with it, such as a bullet or missile would (see Figure 3.10).

You can switch on or off an object's configuration to send or receive collisions at any time (see Figure 3.11). For example, you could set bricks broken off a wall to only receive collisions, so they are pushed around realistically, but don't mess with the player's ability to move. You could also set objects to send and receive collisions or do neither.

Objects can be assigned to layers, as mentioned above, and also to groups. These layers and groups can then be used in collision detection. Let's use a classic

Figure 3.10
Setting up a collision reaction when the fighter jet is hit.

example of a space shooter, where you are leading a squadron of fighter jets against an alien menace. You wouldn't want the squadron of the player's friends to shoot the player's jet. You'd want to turn off friendly fire by assigning objects on the same side and their weapon fire to the same group and then turn off their collision with the player's group.

Physics Engine TGB's physics engine has been modeled after real-world Newtonian physics, to make your games seem even more real. Configuring objects is as easy as enabling automatic collision responses for the object and setting the object's physics parameters, including friction, restitution, relaxation, damping, density, and a force scaling factor. Similar to how collisions work, you can turn on or off the ability to send or receive updates to or from the default physics system. You could also assign these forces to a material datablock, enabling you to create a bunch of different objects made of the same material that will all share the same physical properties.

TGB lets you mount one object to another. When you mount an object, the two objects travel around together, plus mounted objects can have more objects

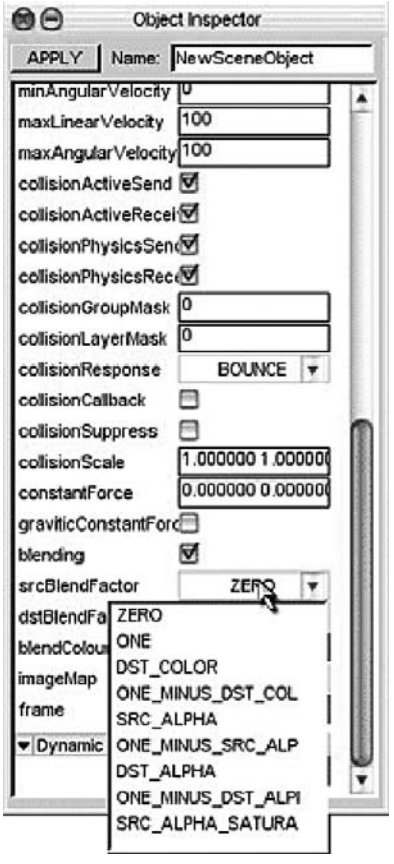


Figure 3.11
Setting up individual collision controls on a new scene object.

mounted to them, thus making it possible to create long chains of mounted objects. In addition, you can specify whether a mount should be rigid (its movement is restricted to that of the parent object), or if it uses *mount forces*. With mount forces, you can dictate that specified forces will be exerted on the mount connection. Thus, a child object could lag behind or be buffeted around relative to its parent.

TorqueNet Lite All Torque games default to a client-server architecture. This delivers more consistent gameplay because a client-server architecture helps prevent hacking. You can also play your game creations with all your friends using TGB's incorporated turn-based multiplayer support system, TorqueNet Lite.

TorqueNet Lite is an event-based network library that is as close to being a real-time ghosting solution without being one. This means that when playing the game, the players don't have to worry quite so much about lag times and being left out of play. Additionally, TGB Pro users can modify the source code to include higher-level networking functionality using resources available on the Torque Developer Network.

The Packaging Utility Write your game on any platform you happen to work from and you can port it to Windows, Mac OS X, or Linux with just a little work. Also, TGB's Packaging Utility makes packing your game up for burning to CD or Internet download trouble-free. The Packaging Utility analyzes your files and saves them in the most efficient manner it can find.

Torque Community TGB also comes with one of the best support communities any indie developer could hope for. The GarageGames web community is top-notch and the knowledgeable staff and individuals driving it can provide you with all the help you need to get started. If you have a wish list or query, take it there. The GarageGames site also provides excellent resources, including scripts, code snippets, links online, reference material, books to help you out, and more. You can find useful forums, tips, tools, and messages from fellow developers.

Other GarageGame Products

The Torque Game Builder is a powerful introduction to game development, but it has its limitations, the most important being that it is preset for 2D games. There are a few other Torque product options that you can try out and see which one works best for your dream game. They include:

- Torque X Engine
- Torque Game Engine
- Torque Game Engine Advanced

Torque X Engine

When Microsoft released their award-winning next-generation console system, the Xbox 360, they also opened the doors for indie game developers to start creating their own games for use with the console. Now you can create your own games for the Xbox 360 and publish them in the Xbox Live Arcade marketplace.

In order to do this, Microsoft released the XNA Game Studio Express December 11, 2006. The XNA Game Studio Express uses C# program code to

design both Xbox 360 console games and Windows PC games. The same day the XNA Game Studio Express came out, GarageGames released a beta of their concept technology, the Torque X Engine. The *Torque X Engine* actually allows users of the Torque Game Builder to make games to port to the Xbox 360, through a WYSIWYG system called the Torque X Builder.

To port games to your Xbox 360 using Torque X, you must be a member of the XNA Creator's Club. The XNA Creator's Club is set up by Microsoft for users of programs to port their games to the proprietary console machine. You can find out more about the XNA Creator's Club online at http://msdn.microsoft.com/xna/creators.

Torque Game Engine

The *Torque Game Engine* (TGE), GarageGames' 3D cousin to the Torque Game Builder, is consistently ranked #1 on the Top 10 Commercial Engines listed on DevMaster.net. Game designers have been proving time and again that TGE works, making the list of games created using TGE an exponentially growing one. Vivendi, NC Soft, and even NASA have developed projects using TGE.

TGE (see Figure 3.12) is a complete proven package for a wide range of applications, with a flexible and controllable in-engine 3D Toolset and award-winning TorqueNet networking. Some of the latest games made with TGE include:

- *Age of Time*
- *Blockland*
- *Buccaneer: The Pursuit of Infamy*
- *Dark Horizons: Lore Invasion*
- *Desert Gunner*
- *Dimenxian*
- *Golden Fairway*
- *Magecraft*
- *Marble Blast Gold*
- *Minigolf Mania*

- *Minions of Mirth*
- *Once Upon A Time*
- *Orbz*
- *PCD Music Lounge*
- *Penny Arcade Adventures: On the Rain-Slick Precipice of Darkness*
- *RocketBowl*
- *Sachi's Quest*
- *Shelled!*
- *Shokrok Throwdown*
- *ThinkTanks*
- *TubeTwist*
- *Ultimate Duck Hunting*
- *Wildlife Tycoon: Venture Africa*

Figure 3.12
The Torque Game Engine logo.

The Torque Game Engine Advanced

If you were an experienced programmer ready to make truly amazing next-generation games with the hottest game technology available, then you would need the next level of Torque. Torque Game Engine Advanced (TGEA) lets you do dramatic visuals and beautiful high-quality effects, on top of an underlying foundation of physics, networking, scripting, animation, and the rest of the game engine—to get your professional game developed in record time to meet your milestones and at a fraction of the cost.

TGEA has support for modern shaders and a much more tailored rendering engine to improve look and efficiency to make it easy for you to create pure eye candy. This engine empowers you to do it all.

Resources

The CD-ROM that comes with this book will provide you with data files to complete the exercises. You can find a downloadable trial demo of the Torque Game Builder software application from the GarageGames' website: http://www.garagegames.com. Although the majority of this text will focus on game creation using TGB, many of the same basic principles apply to game making in general and the lessons will form an early introduction to game design using game engines.

Note

Josh Williams, CEO of GarageGames in 2008, says: "We've worked with developers for years and seen many indie games not reach the sales numbers they deserve. . . We now have an outlet that promotes great games without the complications that typically arise with trying to publish a game: InstantAction. *InstantAction* is both a huge opportunity as an indie developer to create games on your own terms, as well as a great place to play games as a gamer. It's now as simple as hopping on the site and clicking a game to jump right into playing good quality, core action games. *Your* quality, core action games! Each game available on InstantAction thus far is a Torque game: *Marble Blast Online*, *ThinkTanks*, even *Cyclomite*, which was created by Alex Seropian's new studio Wideload. Once you get your games made, share them online at InstantAction at http://www.instantaction.com.

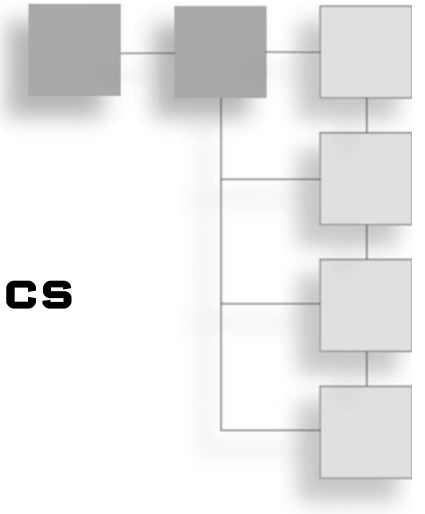
Review

After reading this chapter, you should know:

- How the Torque Game Builder (TGB) got started.
- How to license TGB to make your own games.

- What the major components of Torque are, including 2D Toolset, PixelBlast, and more.
- What the optional editions of Torque are called and what they do.
- What resources this book comes with to help you use Torque to make your own games.

CHAPTER 4



Making Game Graphics

In this chapter, you will learn:

- The definition of digital graphics
- How virtual memory, pixels, and coordinates work
- What 2D graphics are used for in games
- How vector art operates
- The way to create and edit game graphics in an image editor
- How to make static and animated sprites in TGB

This book primarily focuses on game graphics, basic game design, programming, and marketing. Graphics are any visual images you create to be seen. Computer graphics include anything of a visual nature that artists create using the computer as a tool. Computer graphics are also digital. By "digital" we mean that the computer digitizes—or breaks down images into smaller digits the computer can translate. Digital means the computer converts analog data into on/off binary digits in a format that all computers can read.

Whenever you use a digital camera to take pictures, you're not recording directly as an imprint of exposure on film; you are creating a digital code in the camera's onboard processor that allows you to move your pictures on to a computer or print them out using a color printer.

The on/off values the computer uses as binary format is really a sequence of numbers, often billions of numbers. Whenever you take a snapshot with your webcam, you're recording a sequence of numbers that can be used to create a duplicate of the original artwork; or the code can be altered to display a change in the appearance of the image. Digital graphics can be seen in various media and entertainment fields, and so the topic of digital graphics is huge and can get very technical.

This book targets computer graphics as they are used in modern video games. The visuals seen in today's games require artistic talent as well as technical knowledge.

What Are Game Graphics?

Visuals are a huge selling point for games, and it is the stunning graphics that initially draw consumers to them, similar to how special effects draw audiences to box office thrillers. Because looks are so important to making a good first impression, computer graphics is a highly sought-after field of study. Today gamers have learned to look past just the graphics when purchasing a game, but there was a time in the history of video games when some games truly did sell based on looks alone (see Figure 4.1).

Computer graphics is a huge area of study that will continue to grow as time progresses. Game graphics have increased considerably in technical terms, quality, and art style. Every time a new game makes a leap in visual quality, it raises the bar for the entire industry. With games getting better in visual quality, they are gaining ground against the motion picture industry.

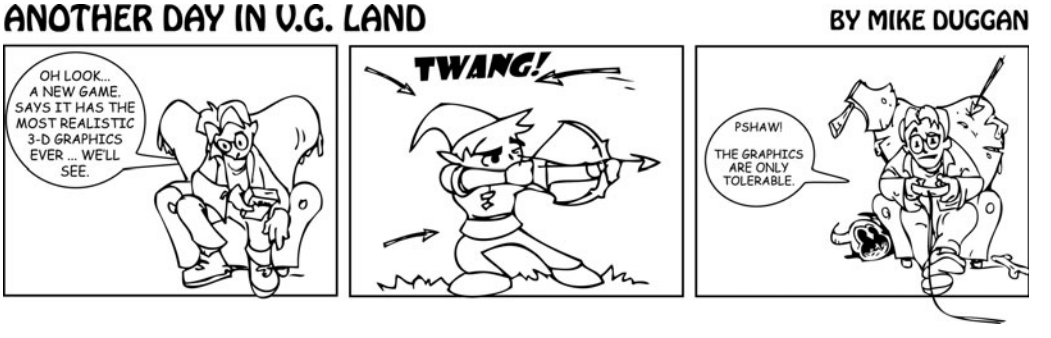


Figure 4.1
Game graphics, some players argue, can never be real enough.

This chapter focuses on computer graphics used in 2D video games. The goal of this chapter is to give you the information essential to producing the type of visuals and effects you see in 2D games. Game graphics are a challenge, and this chapter aims to demystify various techniques and effects standard in video games.

Bits and Bytes

The smallest unit of virtual memory in a computer is the *bit*, which comes from the phrase "binary digit," and a bit is either a 1 or a 0. The next smallest unit of memory is a *byte*, which has eight bits. One bit stores up to two values (0 through 1); two bits store up to four values; three bits store up to eight values; four bits store up to 16 values; five bits store up to 32 values; six bits store up to 64 values; seven bits store up to 128 values; and eight bits store up to 256 values, or 0 through 255. Notice that the number 256 is exactly how many colors there are available in the common computer color palette. Coincidence? Not!

Memory storage has labels based on number of bits. For example, a byte is eight bits, a kilobit is 1,000 bits, a megabit is 1 million bits, a gigabit is 1 billion bits, a terabit is 1 trillion bits, and a petabit is 1 quadrillion bits. The list goes on forever. When you go to buy a computer you'll usually look at memory in terms of bytes rather than bits, with a kilobyte being 1,000 bytes (8,000 bits), a megabyte being 1 million bytes, and so on.

This is all very important when storing, retrieving, and editing graphics, as you'll soon see.

Graphics Pixels

Most of the images you scan into your computer, from either a scanner or digital camera, use a conceptual approach called raster graphics. The raster image you see on your computer is composed of tiny picture elements called *pixels* for short laid out on a rectangular grid. The standard screen may contain over 1.3 million pixels. They cover the screen like a checkerboard, making up countless little squares of color, as seen in Figure 4.2.

The idea of the pixel isn't too far from that of the halftone photos you see in the newspaper. What appears to be tonal variations of gray are, when viewed close-up, really just a bunch of tiny dots, each a different shade. Likewise, pixels are different shades of red, green, and blue (acronymed by tech geeks as RGB). Seen from a distance, a screen image made up of RGB pixels blends together to form continuous tones.

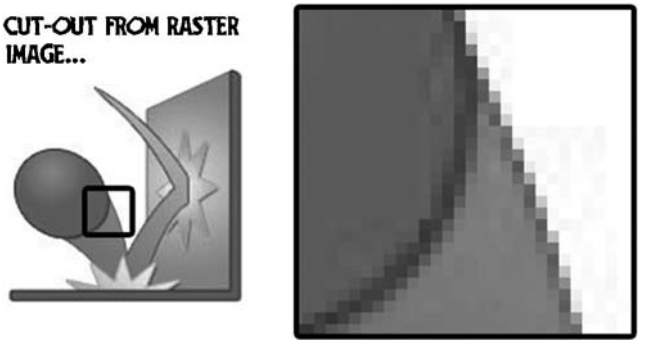


Figure 4.2
An example of pixels.

Each pixel represented in an image is given three numbers to identify its color: one each for red, green, and blue. Plus, people write down three numbers in red, green, and blue in order to save time and space. For instance, you might say that one pixel has a red intensity of 5, a green intensity of 225, and a blue intensity of 10—or 5, 225, 10 for short—meaning that the pixel looks lime green.

As you can imagine, even a small picture can get quite large in computer data with all the code attached to it to cover color information. This is why large color photos can take up so much room on your computer.

Many paint editing software programs can change the visible pixels to enhance the image in many ways. These image editors alter the visible pixels by changing the numbers describing them.

Graphics Coordinates

Each pixel has a position on the screen. You can describe this position by using a number.

Computer screens use a coordinate system similar to a road map of a big city to know where each pixel lies. Going lengthwise across the screen horizontally, much like longitude on a map, is the *X coordinate*. Going heightwise across the screen vertically, much like latitude on a map, is the *Y coordinate*. Using both X and Y, you can find where each pixel is at any given time. For instance, a pixel might be defined as 45, 0—meaning that the pixel rests on the top of the screen about an inch or two over from the left, as seen in Figure 4.3.

It might help you to learn some basic geometry before attempting to study the coordinate system of X and Y on your computer screen. It will come in handy

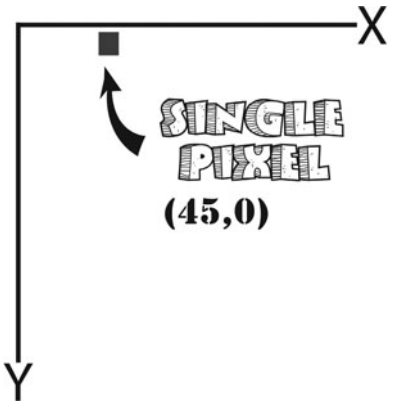


Figure 4.3
X and Y coordinates mapping a pixel on the screen.

when you starting programming trickier parts of your games, because with 2D games you will be using X and Y coordinates over and over again to place graphics, find out where the mouse pointer is pointing, and to determine when certain objects overlap on the screen. It should be noted, however, that sometimes the XY coordinates can appear to change, because—depending on the tool you are using—the origin of the XY grid can start from a different center or a different corner of the computer screen.

Image Types

There are many different image file types out there, but it's safer to remain consistent when working within a pipeline production. That is one reason why it is suggested that as you work with the Torque Game Builder you stick to using one of two major types of files as images: JPEG or PNG.

Joint Photographic Experts Group (**JPEG**) files have an extension of .jpg, .jpe, or .jpeg. A JPEG (pronounced *JAY-peg*) is a graphic image saved using compression techniques to make it smaller for download/upload. When creating a JPEG image, you can specify the balance you want to reach between image quality and file size.

Portable Network Graphics (**PNG**) have a .png extension. The PNG (pronounced *ping*) format also is a compressed file format supporting multiple colors and resolutions. The World Wide Web Consortium, also known as the W3C, developed the PNG format as a graphics standard and patent-free alternative to the GIF format (pronounced *giff* or *jiff*). PNG can also support alpha and palletized formats, which JPEG cannot.

What Are 2D Graphics Good For?

Photos, wallpaper, and pretty stuff to put on your MySpace page notwithstanding, 2D graphics have some very important roles to play in video games, which we'll discuss in more detail now.

Sprites

2D games use sprites for the virtual objects and elements found throughout a video game. A *sprite* is a 2D rectangular image that is drawn to the screen. Whereas in 3D games sprites are used for small things such as particle effects because a 2D depiction of minute entities is more practical for fast rendering, in 2D games sprites are used for everything you see on your computer screen.

There are generally two types of sprites: static and dynamic. A *static sprite* is a single sprite image that consists of a non-animating character, object, or element. A *dynamic sprite* is one that will be animated, like a character walking across the screen or a waving flag on a pole. Although the screenshot is not animated, you can get a general idea by looking at Figure 4.4.

Figure 4.4
The clouds and rocks are static sprites, while the player character (the airship) is a dynamic one.

Tiled Images and Backgrounds

In 2D games sprites are placed together to construct the environments and backgrounds. One or more sprite images act as the environment's background, such as the sky, the clouds, horizon, trees, mountains, or other. This background often scrolls with the player to give the effect of the player moving through the environment, an effect that can be enhanced with parallax scrolling algorithms.

In addition to the background elements, there are often foreground elements. Often this foreground element is made up of tiles. A *tilemap* is any regularly spaced grid that reuses the same set of tiles, and a tile is any sprite image that is composed of a repeatable pattern and can represent a real terrain. By placing a tile-able image next to other tile-able images of the same set, an artist can create simulated seamless environments, as you can see in Figure 4.5.

The Draw Order

In video games, one of the most important pieces of information used during rendering is depth information. Depth information is stored in the *Z buffer* and returns a value that lets the rendering API know the calculated depth of each pixel

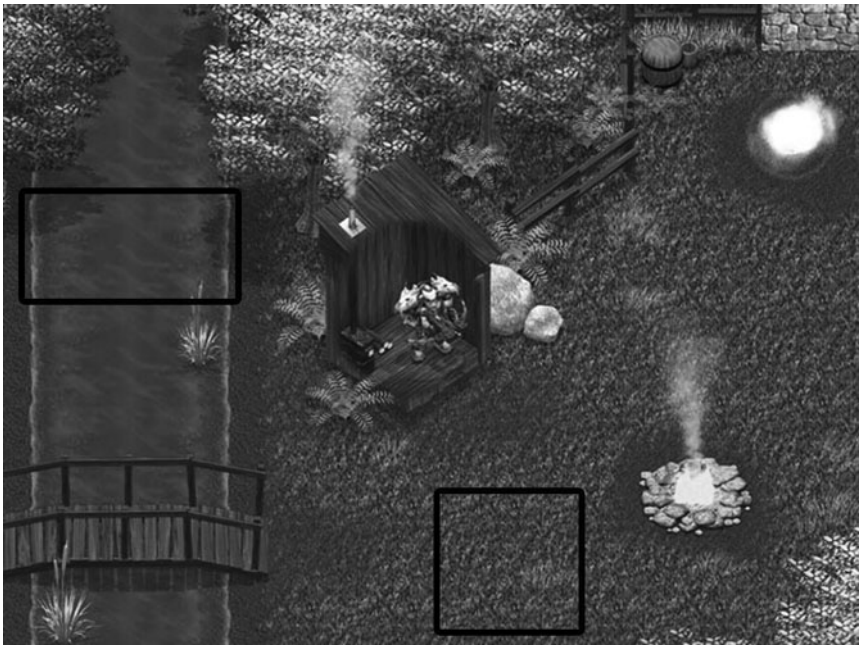


Figure 4.5
The square borders demonstrate tilemaps laid down to represent a grassy terrain and placid creek.

on the screen. This information is crucial to the game's *draw order*, which is used mainly to determine what objects are in front of others, which are behind others, and when one object collides with another.

In 2D games, the sprite images are flat and therefore simple enough they can be given a single value for each sprite and a draw order so that the objects are drawn in the order specified.

Making Vector Art

Now that you have the barest introduction to game graphics and understand the uses of 2D art in games, we'll look at creating your very first game art. For our purposes, we'll create a campy hearse using the vector art program Adobe Illustrator CS3. You can download a 30-day trial version of Adobe Illustrator CS3 from the Adobe website: http://www.adobe.com/products.

Illustrator isn't the only vector program available, either. Another popular software application is CorelDRAW, which has many of the same features and may be of more interest. Also, if you find you like Illustrator but cannot afford to purchase the full version, there is a *free* open source image editor program called Inkscape that has a streamlined interface, and Inkscape has a lot of the same features as Illustrator. You can download Inkscape from the website: http://www.inkscape.org.

Go ahead and install the demo of Adobe Illustrator CS3 for the following lessons, and you can decide whether to use it, Inkscape, or CorelDRAW later.

Of course, you might find that art just isn't your "thing" and you might decide to go into programming, because it *is* your "thing"—and that's fine. Not everyone can be a game artist. But having a broader understanding of what goes into game development will set you head-and-shoulders above the rest.

What Is Vector Art?

There are two major types of artwork when it comes to digital graphics: one is pixel-based (often referred to as raster), and the other is vector-based. See Figure 4.6 for a contrast between pixel-based and vector-based art.

With pixel-based images, designers have control over the appearance of the smallest details, pixel-by-pixel, which is what Adobe Photoshop is so good at, but pixel-based images have a limited amount of detail defined by the pixels per inch

Figure 4.6
The same image (a paintbrush) shown in pixels and in vector.

(ppi) of the image. Enlarging a raster file makes the image look fuzzy around the edges, and the more you magnify it you'll see the blocks of color making up the illusion of continuous tone.

In contrast, vector art is defined by plotting control points and using mathematically defined lines, such as straight lines, Bezier curves, or even NURBS, and can be scaled to any size without loss to detail or quality. As you enlarge the image, the computer does some fast math and plots the points at new coordinates, continuously redrawing the Bezier paths connecting them. Vector art is easier to edit, because the computer only has to keep track of control point coordinates, but you don't have the freedom of changing individual pixels. Adobe Illustrator, as well as other vector art programs, let you mix vector and raster art for better quality work.

What Is Illustrator Good For?

Look around you. Packages of breakfast cereal, billboards on the sides of buildings, gum wrappers, navigation icons and avatars on the web, advertisements filling the spaces of your favorite magazines. . . All of these and so much more have been created by talented designers with the help of Illustrator.

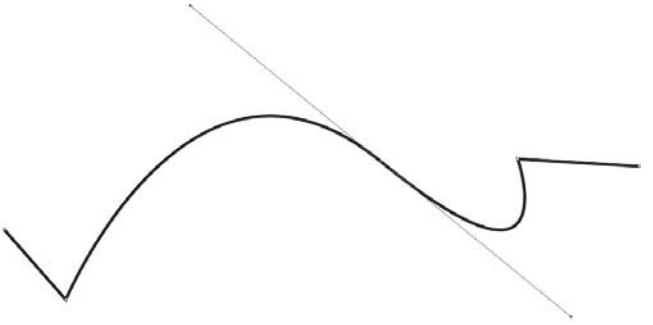


Figure 4.7
An example of Bezier curves.

Illustrator has been used by designers wishing to express their creativity in print, on the web, in video, and on wireless devices. The program is a close cousin of Adobe's other hot image editor, Photoshop, except Illustrator focuses on Scalable Vector Graphics (SVG) file formats. Using Bezier curves, Illustrator can create smooth images of any size and for any output.

Bezier Curves

The neat trick that makes computer-aided drawing possible is the Bezier curve, named for Pierre Bezier, who developed it in the 1970s for drafting purposes. The Bezier curve (see Figure 4.7) lets designers create extremely accurate drawings using complex curves with precision handles.

The designer creates control points along the line that is drawn and uses these control points to control the shape, direction, and amount of the curve desired. Since Bezier curves are vector art, they can be enlarged to any size. Bezier curves are found in software applications like Adobe Illustrator and CorelDRAW.

The Illustrator Environment

Adobe Illustrator, the same as most of Adobe's applications, has many tools, panels, windows, menus, and commands that enable you to get your projects done. Adobe calls this environment your workspace, imagining that the environment within the application is a big desk surface where you work.

At any time during your work, you can open individual documents within Illustrator and manipulate them as separate entities.

Upon launching Adobe Illustrator CS3 for the first time, you'll be greeted by the Welcome Screen (see Figure 4.8). From here, you can choose to open preexisting files or create a new document. New documents can be for print, for the web, for mobile devices, or for video/film. When designing game art, you'll typically

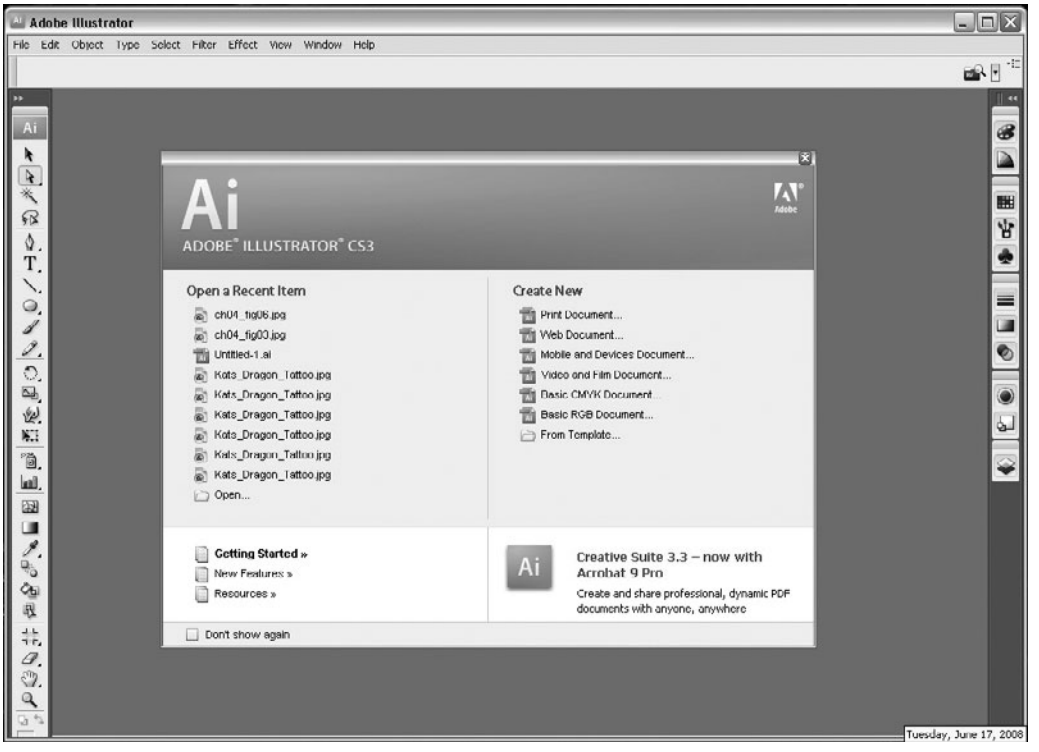


Figure 4.8
The Adobe Illustrator CS3 Welcome Screen.

choose print documents, but make sure that the color selection is set to RGB rather than CMYK.

Note

CMYK refers to the color palette cyan, magenta, yellow, and black and is the gamut (range) favored for printing because it's a subtractive color process. By subtractive, I mean that the base value is set to white and color is added to this to reach desired tonal values. This is very good for print, but RGB is preferred for electronic media such as TV, digital cameras, and video games, because computer monitors read true RGB (red, green, and blue) palettes invariably. For every color in the CMYK spectrum, there is an equal color in the RGB spectrum, and vice versa.

Just as with any software application you choose to work within your pipeline, you have to feel comfortable using it. Adobe applications are generally well known for their customizable workspace, meaning that you can rearrange where all the tools, panels, windows, and so on are so that they fit your individual needs. To know what works best for you, you have to get to know the various menus and tools well and then customize them to fit your workflow.

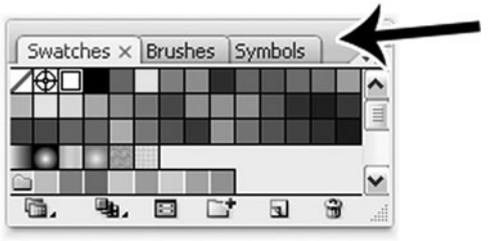


Figure 4.9
Panel name tab.

Illustrator is loaded with tools and panels and menu options. Each panel performs specific operations, and some of them you'll use frequently, such as the Color and Layers panels, and some of them only rarely if at all. Panels can be shown or hidden at will. They are found in alphabetical order under the Window menu found in the main menu at the top of the screen. A check mark next to a panel's name indicates that the panel is open on screen. These panels are called *floating panels*, because they rest above the Illustrator workspace and can be moved in any direction to better fit your needs.

You can reposition panels by clicking and dragging on their name tab (seen in Figure 4.9). You can group several panels together by dragging one panel into the tab part of another one. Or you can attach a panel to the bottom of another by dragging it to the lower section of that panel until you see a blue highlighted line appear and then let go. To avoid confusion of too many floating panels open at once, Illustrator CS3 has what they call *panel docks*, which are organization systems for controlling and minimalizing the open panels. Panel docks (see Figure 4.10) are found to the right of your workspace.

You can choose to view your artwork in multiple ways, each with its own benefits. The default view, *Preview mode*, allows you to create and edit your artwork while viewing it in its final representative state. In *Outline mode*, Illustrator hides the fill colors and shows you only the outline geometry making up your shapes. Outline is great when you're first making your initial shapes with Bezier curves, so that the fill colors don't cover up where you want to go. You can toggle back and forth between Preview and Outline mode by pressing Ctrl+Y (Command+Y). See Figure 4.11 for a contrast between the two modes.

Although you can scale Illustrator artwork to any size without losing its crisp look (SVG-capable applications call this being "resolution-independent" because the artwork is not dependent on the screen resolution of the user's

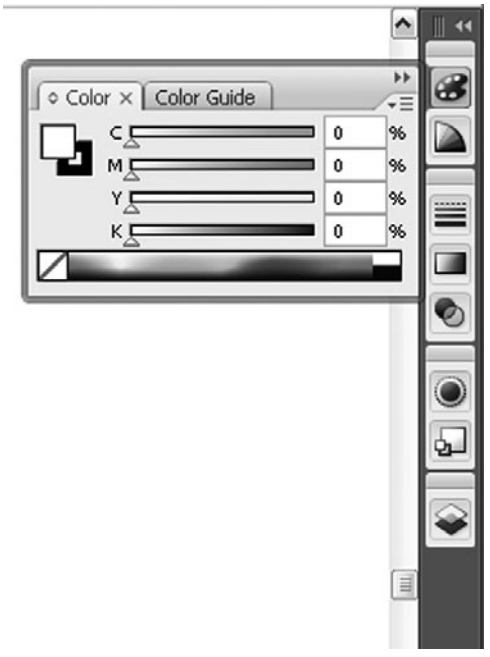


Figure 4.10
Panel docks.

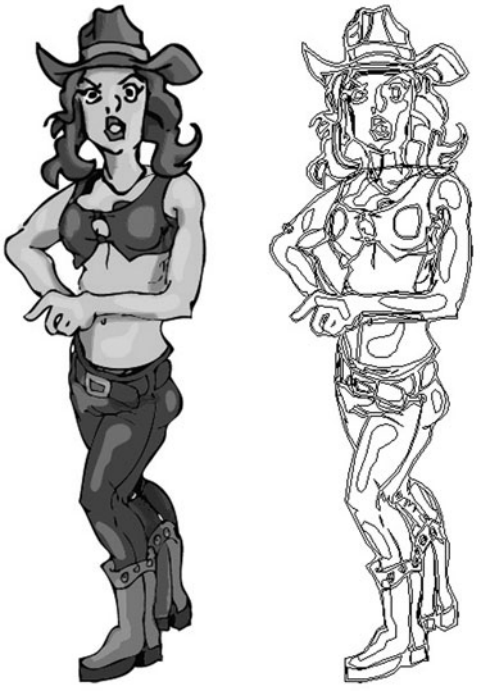


Figure 4.11
The same image of a cowgirl shown in Preview and then in Outline mode.

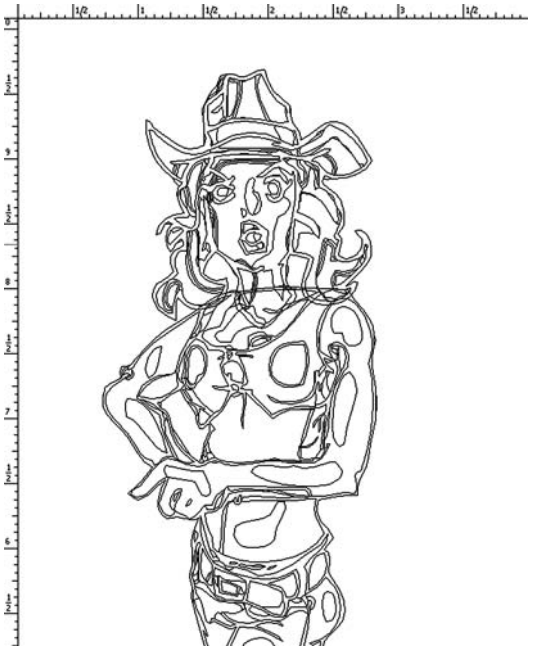


Figure 4.12
The rulers in Illustrator.

computer to be viewed), it's also important to take a moment and look at measuring elements on the Illustrator artboard.

Choose View > Show Rulers to display both up-and-down rulers that appear along the left and top edges of your document window (see Figure 4.12). You can Ctrl+-click (right-click) on a ruler to change its measurement system. The most general method used for measuring in Illustrator is in inches, unless you prefer the metric system. If you want to be really technical or you're working for a major web design firm or printing/publishing company, you might want to switch to picas or pixels. For now, choose inches.

Rulers are good for measuring when you're making shapes on your artwork, but they also serve to make guides. A *guide* is any line that's faintly visible on your screen but won't be printed or exported with your artwork. You can click any ruler and drag a guide out onto your artboard. Guides can be positioned anywhere you want them to, and when you have them where you want them you can choose View > Guides > Lock Guides to prevent them from being nudged by accident.

Figure 4.13
An open path (line) and a closed path (rectangle).

The Anatomy of Vector Objects

Before getting too hot-and-heavy making your first vector art for a game, it's important to cover some basic terminology and get your feet wet playing with the tools in Illustrator. The only way you'll learn a software program is by experimenting with it (books only take you so far). So the following is a quick guide for vector objects and their makeup.

Paths and Anchor Points

A single line drawn on the artboard has two plotted anchor points and a straight line running between them. This is called an *open path*. But if you wanted to make a rectangle that you could fill with any color you like, you'd need to make a closed path. A *closed path* is any path that begins at one anchor point and then returns to that same anchor point. See Figure 4.13 for a comparison.

The corners of a rectangle are straight corners, and the anchor points making up the straight corners are called *corner anchor points*. Anchor points can be straight or curved, however. Anchor points that are connected to each other by curved lines are called *smooth anchor points*, and they have special attributes like the directional handles, which specify how the curved lines are drawn. The positions of these *directional handles*, which have control points on the ends of them as well, define the curve (see Figure 4.14 for an example).

Confused yet? Don't be! The good news is that, for the most part, Illustrator has primitive drawing tools that allow you to create simple shapes without having to worry about anchor points and directional handles. When we get into the lesson, you'll use the Pen tool and you'll get all the experience you need with these attributes. For now, use the primitive drawing tools to play around with the software and see how easy it is.

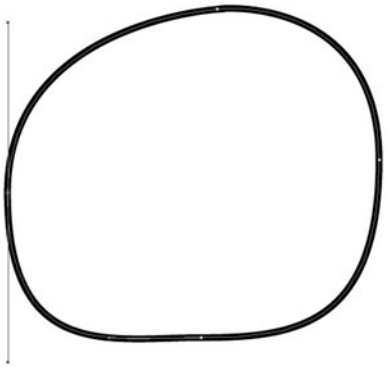


Figure 4.14
Directional handles make up this oval.

Figure 4.15
The closed-path tools in Illustrator.

The closed-path tools in Illustrator include the Rectangle, Rounded Rectangle, Ellipse, Polygon, Star, and Flare tools (as seen in Figure 4.15). Test-drive each of them to see how they work. You can hold down the Alt (Option) key to draw the shape out from its center point rather than from the corner. You can hold down the Shift key to constrain all sides to be equal, resulting in a perfect square, circle, or parallel polygon or star.

Fills and Strokes

The anchor points and directional handles make up the physical look of vector art, but there are two other attributes to every object you should be aware of: the fill and stroke. The *fill* of a vector object is the appearance of the area enclosed by the path; in other words, the interior space. The *stroke* of a vector object is the appearance of the path or outline itself; in other words, the borders.

Fills can include solid colors, gradients, or patterns. These can be found in the Swatches panel, which includes tiny thumbnails of them along with their names, and you can apply a fill by selecting the object you want filled and clicking on one of the thumbnails.

Figure 4.16
The fill and stroke icons.

The Tools and Color panels each contain two overlapping square icons that represent fill and stroke (see Figure 4.16). Pressing the X key on your keyboard will toggle the application's focus between the fill and stroke. Depending on which one is active, selecting thumbnails from the Swatches panel will add the swatch to the fill or stroke.

You can also create your own swatches. After adjusting a colored fill or stroke in the Color panel, you can drag the thumbnail icon of your customized work into the Swatches panel, then double-click the new swatch thumbnail to give it a custom name. This will help you as you progress, in case you need to go back and use a colored fill/stroke on more than one object.

Later, we'll look at another way of coloring our handiwork: using LivePaint. LivePaint is faster than selecting objects and choosing fill/stroke swatches individually, and it's a great way to play around with your vector art to get the desired appearance.

Transforms

Illustrator has several transform tools that enable you to manipulate the final appearance of a graphic after creating it. They include the Rotate, Scale, Reflect, Shear, and Free Transform tools (as seen in Figure 4.17). The functions of the Rotate and Scale tools are self-explanatory: Rotate allows you to rotate the graphic, and Scale allows you to change the size of the graphic. The Reflect tool "flips" an object across an imagined axis, horizontally or vertically. The Shear tool slants an object across a specified axis, distorting the object. Last, the Free Transform tool offers you the ability to perform quick transformations and distort objects in perspective.

Figure 4.17
The transform tools in Illustrator.

Each of these tools is easier to use through the tool's dialog box. Double-click the Transform tool to open the tool's dialog box. Enter the values by which you want to execute the transformation, then click OK. You may also click on Copy to create a transformed duplicate of the original object. If you like the transformation and want to do it again immediately afterward, you can use the Transform Again command by pressing Alt+D (Option-D).

Making a Hearse

Getting started with drawing an illustration is often the hardest point. How do you get from what's in your head to a vector object?

Sometimes your illustration is just an idea rolling around in your head, and sometimes it's an image from a photograph or sketch. Drawing an illustration from scratch means starting a blank Illustrator document and using only the Illustrator tools. This approach is widespread, especially when you're drawing something simple. Many talented artists are able to create complex graphics in Illustrator "off the cuff," and they can be amazing to sit and watch work.

I don't expect you to be a master at SVG files at this point, so the preliminary sketches have been made for you and you'll use a trace method.

Using the Place command, it's easy to import scanned images into Illustrator. Tracing a scanned image is not "cheating." An innovative drawing is an innovative drawing, whether it's first fashioned on the computer or drawn on a piece of paper and scanned in.

Drawing the Hearse Parts

What follows is a lesson that will introduce you to the tools, menus, and panels found in Illustrator, and you'll be designing a hearse graphic you can use in your video games. The main importance of this lesson is to learn the Pen tool. To master Illustrator (and to become a better designer) you must master the Pen tool.

1. Open C4-1.ai from the data files (found on the CD-ROM) and save it as Hearse Parts.

2. Click View on the menu bar, then click on HEARSE BODY. You should see the sketch outline of the hearse body on your screen (as seen in Figure 4.18).

3. Verify that the fill color is set to [None] (the red hash mark) and the stroke color is set to Black. In the Stroke panel (the Stroke panel dock looks like a

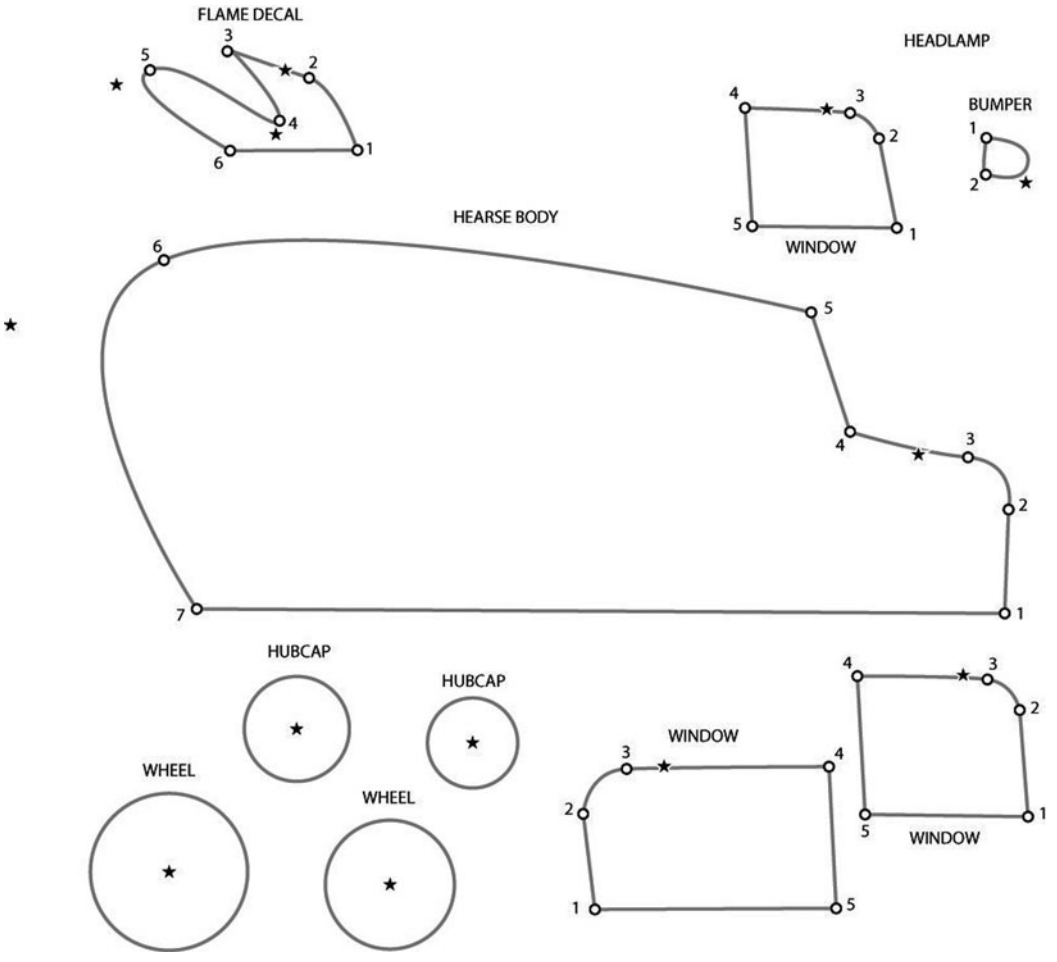


Figure 4.18
The guide sketch of the hearse body.

number of short lines, or you can find it by going to Window > Stroke in the menu) verify that Stroke is set to 1 pt.

4. Click the Pen tool, position it over point 1, and then click on point 2, making a short straight line. Then go to point 3 and drag a direction line to the star on the left side of the 3.

5. Click to 4 and 5, which are corner points, then go to point 6, where you want to click and drag a direction line to the star way out to the left. You are creating smooth anchor points as you go, and using Bezier curves to create curved lines.

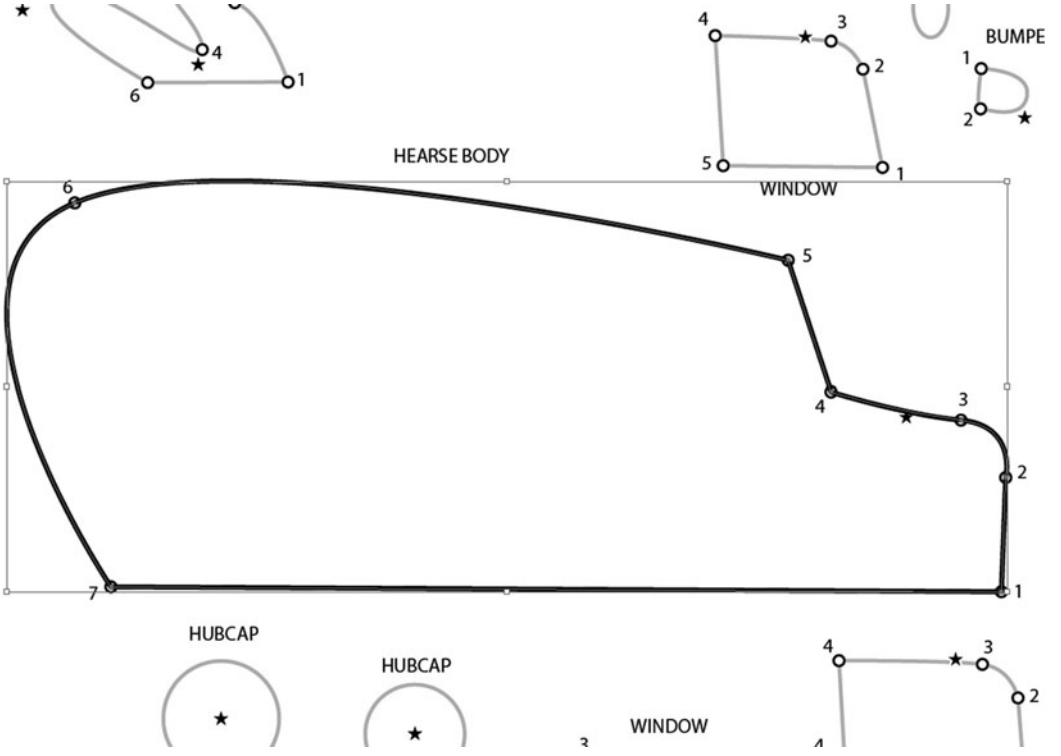


Figure 4.19
The completed body shape of the hearse.

6. Using this same method, complete the drawing. At point 7 you won't need to drag a direction line. Go back to the start, at point 1, and when you see a small circle appear next to your cursor, click once to finish the shape. Compare your screen to Figure 4.19.

7. You've just drawn the body of the hearse. Under View on the menu bar, you'll see that each group of hearse parts has its own view window. Draw the remaining shapes of the illustration, referring to the sketch for help.

8. When it comes to the wheels and hubcaps of the hearse, you should draw the shapes with the closed-path Ellipse tool. Place your cursor on the X in the center of each circle, and holding down Alt (Option) and Shift, click-drag a perfect circle out from the center of the shape to fit the recommended size.

9. Save your work after you complete each element so you don't inadvertently lose anything.

Painting the Hearse

Now you're ready to add attributes to your elements. One by one, you're going to add fills and strokes to the hearse parts. You'll do this with Live Paint, a revolutionary new mode in Illustrator CS3.

The Live Paint Bucket tool uses two new Illustrator object types called *regions* and *edges*. Regions and edges are comparable to fills and strokes, but they are in "live" mode. Where two regions overlap, a third region is created and can be painted its own color. Where two edges overlap, a third edge is created. It too can be painted its own color.

Adobe likes to say that Live Paint is an intuitive working mode. This is because something that looks like it could be filled with its own color can indeed be filled with its own color. As long as you have the Live Paint Bucket tool selected, objects that you have selected can be filled using the new rules of Live Paint mode, and when you exit Live Paint mode, the paint you've applied to the graphic remains as part of the illustration.

Let's try it out now to see what I mean.

1. Verify that nothing is selected on the artboard.
2. Select all by Ctrl+A (Command-A) and choose Object > Live Paint > Make from the main menu. This turns your selected art objects into a Live Paint canvas.
3. Pick the Live Paint Bucket tool from the tools panel, if you haven't already. Notice that your mouse pointer changes.
4. Double-click the Live Paint Bucket tool in the tools panel to open the Live Paint Bucket options dialog box. Make sure that Paint Fills and Paint Strokes is turned on, so that you can paint both regions and edges seamlessly.
5. Experiment. Move your mouse pointer over an edge to see the pointer change to a brush icon and the edge become highlighted. Move your mouse pointer over a region to see the pointer change to a bucket icon and the region become highlighted. The highlight shows you what you will be painting when you click your mouse.
6. Paint your hearse any colors you like.

7. When you're through, choose Object > Live Paint > Expand from the main menu to exit the Live Paint mode and return to standard editing mode. Once there, you'll have to select the hearse parts and right-click and select Ungroup, until you have separate parts that you can move around again. Be forewarned: what used to be strokes are now complete shapes of their own, and you'll find it easier to use marquee-selection to grab both the shape and the object that used to be its stroke.

Assembling and Exporting the Finished Hearse

Each part of the hearse was a single closed shape, and now you've painted each shape. Next comes the fun part. Just like putting a jigsaw puzzle together, you need to click-drag each shape on top of the other to assemble the finished hearse, just like you see in Figure 4.20.

Of course, depending on when you made each of the parts, you'll notice that the shapes tend to overlap each other or hide behind each other when you don't want them to; this is because every element you created was placed into a different stacking order. You can always change the stacking order of your shapes. With a shape like the hearse wheel selected, right-click on it and choose Arrange > Bring to Front from the pop-up list menu. This brings the selected shape to the foremost, or top, layer in the stacking order. You can also select Send to Back, which pushes the object to the bottom layer of the stacking order, and so on. Arrange each shape in the stacking order as you desire them to appear, and assemble them on your artboard so that the final product looks just like Figure 4.20.

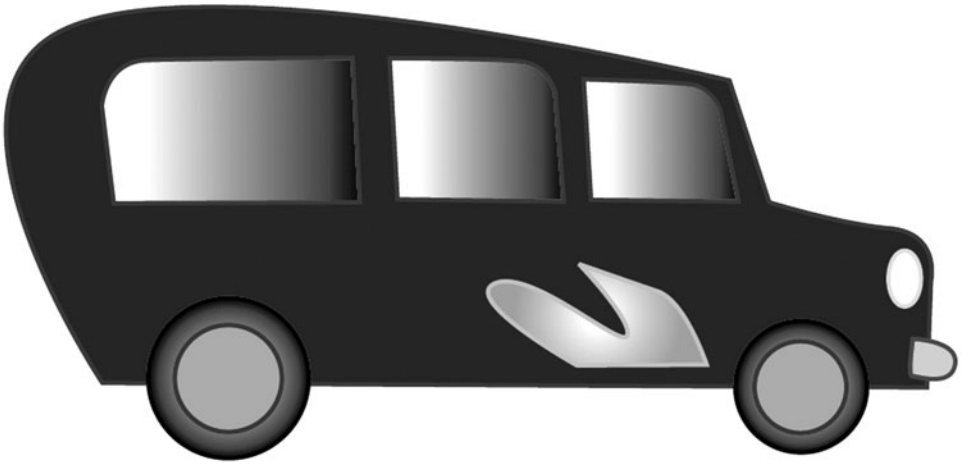


Figure 4.20
The assembled hearse.

Once the project is completed, you are ready to save it for use in your game:

1. First, select the entire hearse illustration by pressing Ctrl+A (Command-A) and Ctrl+C (Command-C) to copy the artwork.

2. Open a new document. It doesn't matter what shape or size the new document is, or what its intention is, because we're only using it as a stage for the next step.

3. Press Ctrl+V (Command-V) to paste your artwork into the new document.

4. Go to File > Document Setup and resize your image to 328 (width) by 180 (height) pixels.

5. Go to File > Export on the main menu to open your Export dialog window. Recommended image file types for use with the Torque Game Builder include JPG or PNG. For full use of alpha transparency, it's recommended to save as a PNG and you can transform it into a JPG later as needed.

As long as you remember that Torque Game Builder prefers JPG or PNG image files, you can use the skills you've learned here to work in any image editing software application you like to create great 2D graphics to put into your games.

Using Graphics in Torque Game Builder

In case you don't remember, a sprite is a two-dimensional graphic image used in a video game. Sprites can be animated, but if they're not they are referred to as static sprites. Sprites are based on an image map, which can be any image file you can import into your game engine. As far as image files the Torque Game Builder engine can use, stick with JPG or PNG files. For most of the resource files we'll be using later on, I have saved them as PNG files, because PNG supports alpha and palettized formats, where JPG doesn't. This means it's easier for me to save a graphic with a transparent background around it; in other words, as a PNG file. Another form of graphic used in Torque Game Builder is the tilemap, which is a specific type of sprite used for laying out levels quickly.

Making Static Sprites

The following shows you the basic steps it takes to turn an image file into a static sprite inside the Torque Game Builder. For the purposes of experimenting, you can follow along by creating a dummy project. To do so, start TGB and choose New Project from the file menu. Name your new project "Goofing01" and use

the Empty Game project template. This will create a new folder called Goofing01 in your Games sub-folder where you have TGB installed.

You should put any image files you want to add to your game in the Data\Images folder of your main game folder, which in this case would be Goofing01. The Data\Images folder will be the repository for image files you plan to use in your game, and there's one in every new project folder, regardless of what you name the project. For now you can copy and paste your hearse image file into Goofing01\Data\Images, unless you have another sprite image you'd like to play with.

To Load an Image File

The following are steps you'll use to load images in the Torque Game Builder.

1. Click the Create tab on the right-hand side of the editor.
2. Click the Create a new ImageMap button under the tab.
3. Locate the folder where you placed your image, which should be your project's Data\Images folder.
4. Simply click OK and your image will be a sprite available in your editor in the Static Sprites section under the Create tab.
5. To place the sprite into your game, simply drag-and-drop the image from the Create tab into the Scene View.

To Set Properties for a Static Sprite

To change the static sprite's properties, select it in the Scene view and click the Edit tab on the right-hand side of the screen. There are many property sections here, each of which can be collapsed or expanded by clicking on the arrows to the left of their names. The static sprite properties you will want to focus on here are under Static Sprite and Scene Object, so expand them.

In the Static Sprite section, you can change the image map. An image map, if you recall, can be any image file you choose to load into your video game.

In the Scene Object section, you can change the following properties for the selected sprite. Simply enter a new or random value and press Enter to see your changes occur automatically.

- **Position X and Y**: These are the coordinates for the sprite's position onscreen.

- **Size, Width, and Height**: These define the size of the sprite. The size changes whenever you click and drag the handles, or the little blue squares, on the selection box enveloping your sprite.
- **Rotation**: This defines the angle of the object. You can also rotate the sprite by holding down on the Alt key while dragging any of the sprite's selection box handles.
- **Auto Rotation**: This is a little trickier. Auto rotation defines the sprite's in-game rotation in degrees per second and only works when the game is running.
- **Flip Horizontal**: Allows you to flip the sprite from left to right.
- **Flip Vertical**: Allows you to flip the sprite from top to bottom.
- **Layer**: Sets the sprite's screen layer. The higher the number entered, the farther away from the camera the sprite will go.
- **Group**: You can use this to set the sprite within a game group, which is perfect for coding harder stuff but nothing we'll go into right now.
- **Forward/Back**: These buttons move a sprite up or down within its own layer, affecting the draw order of objects on the same layer.
- **Visible**: Permits whether the sprite is visible to the viewer or not.
- **Lifetime**: You use this to define how many seconds of screen time an object has before it disappears from the game world completely. If you put in a value of zero, the sprite won't ever be deleted.

Making Animated Sprites

An animated sprite is not a single framed image but a sequence of image frames that, when loaded into a game, appears to move just like in a cartoon. To design an animated sprite, you must make not just one picture but several—as many as it takes to complete the animation cycle. You must also make sure that each frame of animation has identical height and width to every other one. You can do this in Adobe Illustrator by setting Guides equivalent to where the frame edges are and positioning your drawings within these frames.

To look at an example of how an animated sprite should appear, go to the Games\TutorialBase\Game\Data\Images folder where you installed TGB on your

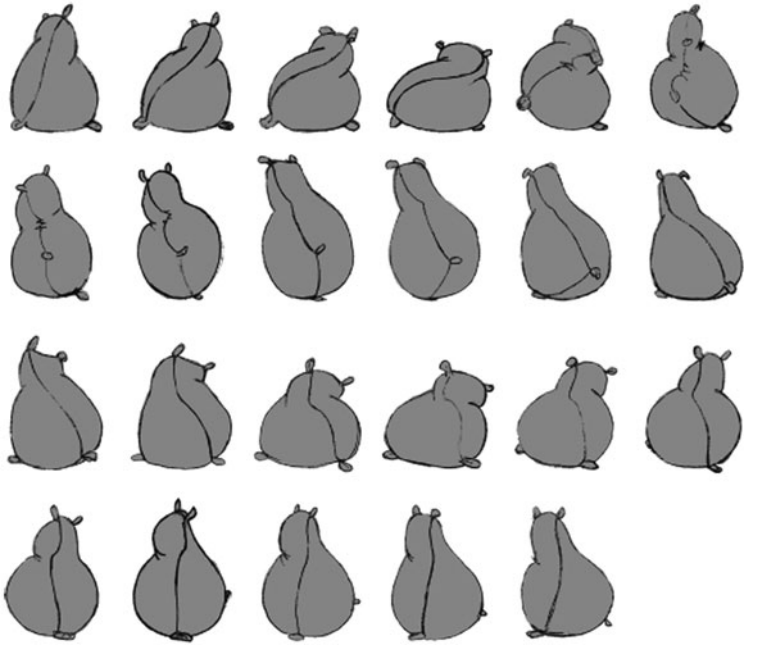


Figure 4.21
Some magician must have made this flour sack get up and walk, rather than carry it himself!

computer. If you cannot see the images, and you are using Microsoft Windows, go to View > Filmstrip. You will then be able to preview the images without opening them. Scroll until you locate the FlourSack.png file (as seen in Figure 4.21). This shows the flour sack character walking, frame by frame.

For the purposes of experimenting, you can create another dummy project. To do so, start TGB and choose New Project from the File menu. Name your new project "Goofing02" and use the Empty Game project template. This will create a new folder called Goofing02 in your Games sub-folder where you have TGB installed. Copy and paste the FlourSack.png file found in the Tutorial-Base\Game\Data\Images folder to your Goofing02\Game\Data\Images folder.

To Load an Animated Image File

The following are steps you'll use to load animated images in the Torque Game Builder. You can follow along by using the FlourSack.png file.

1. Click the Create tab on the right-hand side of the editor.
2. Click the Create a new ImageMap button under the tab. This will open the Choose Your Image File dialog window.

3. Locate the FlourSack.png file, which should be in your project's Data\Images folder.

4. Click the OK button and the flour sack will become a sprite in the Static Sprites section.

5. Double-click on the flour sack thumbnail image under Static Sprites to open the Image Builder. We want to animate this flour sack sprite, so change the flour sack's Image Mode to CELL. Cell mode is used for images that need to be evenly divided into more than one frame. Save when you're through.

6. Click the Create a new Animation button under the Create tab and select `flourSackImageMap` as your source. In the Animation Builder, click the "Add all frames from the image" green plus sign button. Press the Play button on the right to preview your animation of the flour sack in the upper-right corner window. You can adjust the Frames Per Second to another value to speed up or slow down the animation. Once you find a Frames Per Second you see as more suitable, click the Save button and your new animation will appear under the Create tab in the Animated Sprites section.

7. To place the sprite into your game, simply drag-and-drop the image from the Create tab into the Scene view. To see your flour sack walk, select him in the scene, click the Edit tab on the right-hand side of the screen, and expand the Physics section under the Edit tab. Type in the value "**10**" into the Velocity X property field and press Enter. Save your level by clicking on Save As in the main menu, then click the Play Level button in the toolbar. You'll see the flour sack walk across the screen.

Making Tilemaps

Using a tilemap to create repeating elements has several advantages, foremost being time and resources. First, using tilemaps makes level construction faster, saving you time building. Initial level construction can go that much quicker for you when you use tilemaps. Secondly, the engine does not have to waste its resources calling up and drawing a whole image map for a background, but instead only calls up and draws several small seamless ones repeatedly to cover the area.

You must construct your tilemap image files using an image editing program. Tiles have to follow a simple set of rules: they must be square and of a power of 2 (such as pixel size 128 × 128), and they must be seamless. You can lay out these

image files easier in Illustrator if you use Guides to position them exactly. To make each tile seamless, you can duplicate the image and flip its duplicate, comparing edges and dragging their joints until they are centered inside the Guides before deleting the excess. This is a difficult process, however, and one that might take some practice with the software to get right. Or you could try a third-party application that is used primarily for creating seamless tiles, such as those found on http://search.brothersoft.com/tile-seamless.

To Load a Tilemap

The following are steps you'll use to load tilemaps in the Torque Game Builder.

1. In the Level Builder, click on Create a new ImageMap under the Create tab in the sidebar.

2. Locate your tilemap image file. For this exercise, you can use the tileMap.png found in the Games\TGB\Data\Images folder.

3. In the Image Builder dialog window, change the Image Mode to CELL. Set your cell width and height parameters to match the size of your individual textures, which for tileMap.png is equal to 128 (its tiles are 128×128).

4. Click Save.

To Edit a Tilemap

The following are steps you'll use to edit a new or preexisting tilemap in the Torque Game Builder.

1. Select the Create tab on the sidebar, go to the Tilemaps rollout, and drag the object labeled newLayer.lyr into your scene, where it will appear as a block of "test textures."

2. To paint tiles onto your new tilemap or an existing tilemap and set its various properties, you must use the tilemap editing tool. You automatically enter the tilemap editor whenever you drag-and-drop a new tilemap to the Scene View. Otherwise you have to either click on the Edit Tile Layer icon in the edit panel or the pop-up Edit This Tilemap option that appears when you hover your cursor over a selected tilemap.

3. Notice that a Tile Editing rollout will appear under the Edit tab in the Level Builder when in the tilemap editor. This is a special Level Builder mode

unlike others, much like defining world limits or collision boxes around objects. In this mode, you can only work on the currently selected tilemap.

4. Use the tools in the Tile Editing rollout to manipulate the tilemap. They are as follows:
   - **Selection Tool**: Click a tile to select it; double-click a tile to select all connected tiles like it.
   - **Paint Tool**: Applies the settings you choose to the selected tile.
   - **Flood Fill**: This works similar to the Paint tool, but it applies the settings you choose to the tile you click on and to any tile connected that are like it.
   - **Eye Dropper**: Click a tile to copy all of the tile's settings to the Paint tool's settings.
   - **Eraser**: Click any tile to clear all its settings.

   The settings that can be applied to each tile are as follows:
   - **Image**: The image map or animation the tile displays. For the purposes of this exercise, set this to `TileMapImageMap` and use the Paint tool to paint the image in the Scene View to see what it looks like (as seen in Figure 4.22); you can delete the object when you're done if you decide you don't want to keep it.
   - **Tile Script**: The text you want called dynamically in-game when the image first displays onscreen.
   - **Custom Data**: Stores whatever information about the tile you want.
   - **Flip Horizontal/Vertical**: With either or both of these checked, the image on the tile will be flipped.
   - **Collisions**: Enables collision detection on the selected tile, so that if you have a body of water, for example, you can set a collision polygon around the edges so the player character cannot walk on it.
5. In order to exit the Tile Editing mode, you must click the Selection Tool on the main toolbar at the top of the Level Builder.

Review

After reading this chapter, you should understand the following:

- What game graphics are composed of (bits, bytes, pixels, and graphic coordinates)

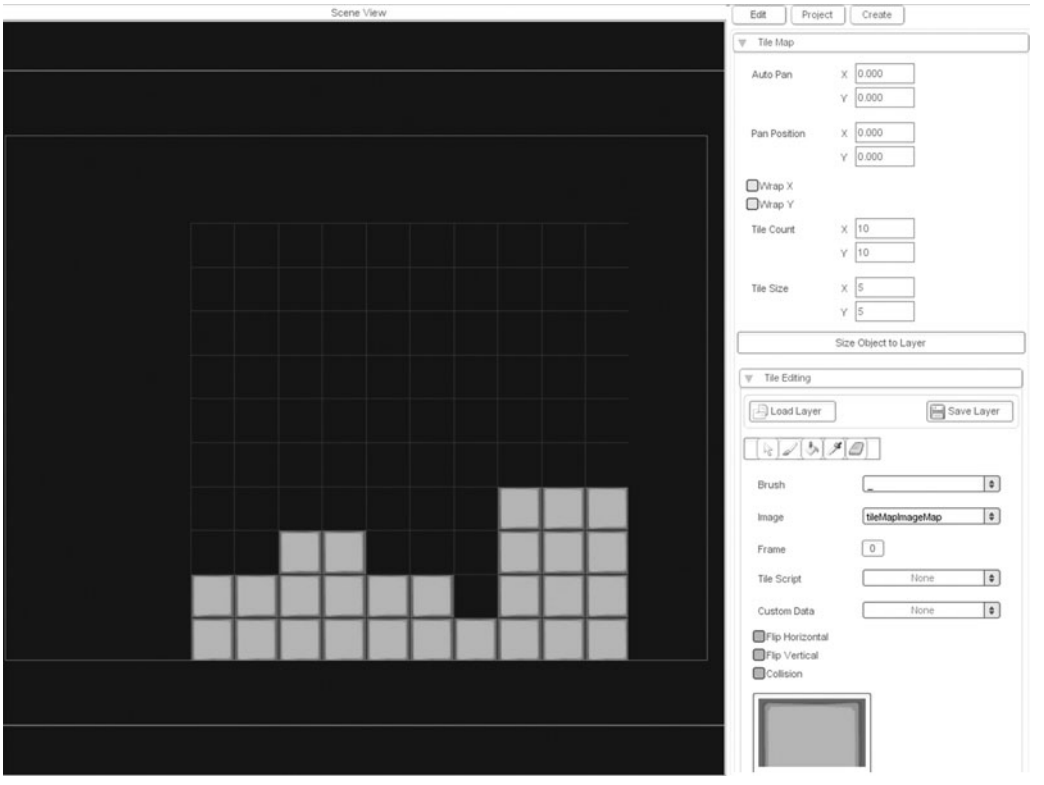
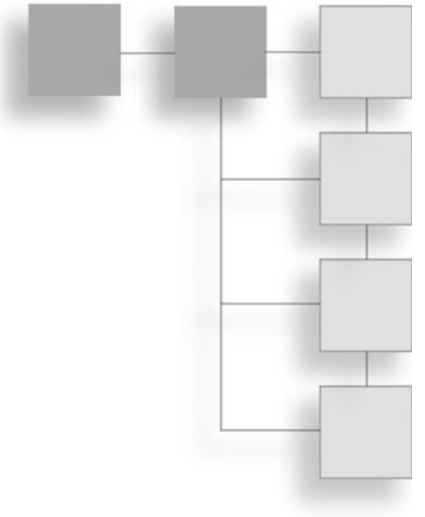


Figure 4.22
Painting a single tile in a new tilemap.

- What 2D graphics are used for in video games
- What vector art is and how to make and edit vector objects in Adobe Illustrator CS3
- How to create an image to use in your very own games and load it in the TGB editor

CHAPTER 5



Programming with TorqueScript

In this chapter, you will learn:

- To make games using programming language
- The features of the TorqueScript language
- What and how to syntax proper algorithms in TorqueScript
- To design programming flowcharts
- What Torsion is and how it can help your development

Graphics aren't everything. Some debaters will say it has been to the industry's detriment that graphics have become the mainstay of popular video games, that nifty graphics (whether done in 2D or 3D) do not make up for lack of content, powerful gameplay, or entertainment. Long before graphics became the primary focus of these games, they existed due to the wonders of computer programming. Taking little bits of an apparently alien language and putting them together to make things appear on a computer monitor is the life-force of any video game. There wouldn't be a video game around today if it weren't for programming!

I know programming might sound difficult. Programming entails mathematics, motion physics, geometry, and sciences you might not be geared up for, but you won't have to worry about it if you want to make a simple and straightforward little game in Torque Game Builder, because you will have the user-friendly power of TorqueScript on your side.

In this chapter, you'll look at game programming, computer logic, and the TorqueScript language you'll be using. You will also learn how to put together simple programming algorithms and flowchart them.

Game Programming

Human-machine interfacing, or HMI, is the way in which a person, or user, interacts with a machine or special device, such as a computer. This user interface requires two things: input and output, or I/O. Most examples of input include the use of keyboard and mouse. Most examples of output include the visual images displayed on a computer monitor and the sounds coming out of the speakers.

To think of using a programming language as input in HMI is similar to comparing a hammer to constructing a three-story building; simple input is nowhere near as complicated as programming is by default! However, you use program code to convince computers to do what you want them to, and computers must interpret your code and spit out what they think you want, which is a kind of output.

How's this for an analogy? Imagine that the programmer is you, and the computer is your race car. Now imagine you're driving your race car (which, unless you have a driver's permit, doesn't give you the permission to try this at home!) and you see a curve in the road coming up ahead. You want to tell the car to slow down and the front wheels to turn, so that you can maneuver the curve carefully. To operate your vehicle, you use a steering wheel and brake pedals. You have gas pedals as well, but we certainly *don't* want to pump those when approaching a hair-bending curve!

Now if the racecar were a computer, and you a programmer instead of a driver, you'd use programming language to tell the computer what you want it to do. This type of interface operates the same way on a driver/car level as it does on a programmer/computer level.

It might be nice if we all lived in a science fiction universe where we all had direct neural implants that allowed us to "think" what we wanted at computers (a scary realization getting closer to reality, as shown in Figure 5.1), but for now that is still fantasy, and the truth remains that we must attempt to talk to computers using a go-between.

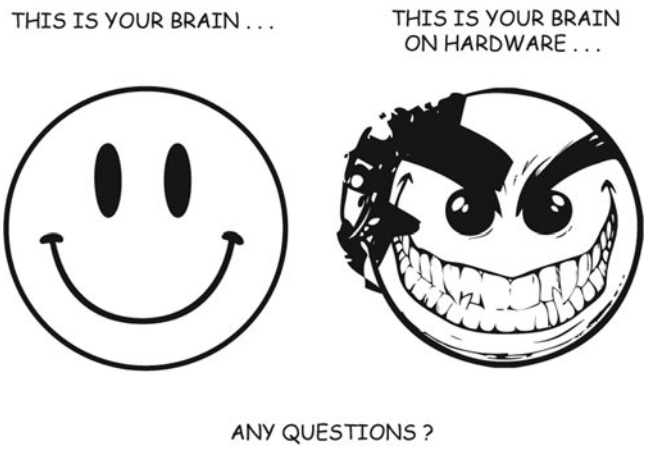


Figure 5.1
A cold shudder is felt when looking at the possibilities of neural networking and becoming a part of a machine, but in actuality neural implants are closer (and less scary!) than you might think.

You see, computers don't "think" in words anywhere near the English language. They communicate in a binary machine code of 1s and 0s, a numeral language that's really just a positional notation with a radix of 2, or switch-on and switch-off, which seems alien to most of us. Most of us are *not* Neo from *The Matrix* movie, so we can't "see" our world in binary, as you see in Figure 5.2. The most

Figure 5.2
Imagine gaining the ability to see your world in binary form. A programmer's ideal fantasy, to tell you the truth!

popular go-between, or translator, we have to talk with computers comes in the form of programming languages.

Programming Languages

Tip

> "I wrote my first game on a remote terminal using APL/360 as the language. My first microcomputer game emulated a complete tank war game on a home-brew system I built that had a whopping 4K of memory and a 512-byte operating system. The point is, a good writer need not blame his tools. He'll make do!"
>
> —Scott Adams

Programming by itself can be tough to learn, but thankfully today you are *not* expected to write in binary code, a hassle that typically scared away most would-be programmers a couple of decades ago! Today most game programmers use specific programming languages, such as TorqueScript, to get the job done.

Programming languages are functional, imperative, and object-oriented (class-based) coding languages used to develop software applications. Because these programming languages are closer to the English language, they are easier for us to learn to program with.

Programming languages use preexisting engine elements to accomplish new tasks quickly and efficiently. In this chapter, we'll look at furthering your understanding by designing and programming your first game project. To do this, you must first recognize the particulars of TorqueScript.

TorqueScript

TorqueScript should look familiar to you if you've ever done any programming before in the C/C++ languages. Most of the *syntax*, or applied linguistics, is the same. A game script is composed of statements, object definitions, function declarations, and packages.

Here are just a few of the most prominent features of TorqueScript that you should memorize:

- **Case-sensitivity**: Variables and functions don't have to be proper case to work. For instance, `%Aargh` and `%aargh` would be read the same.

- **Ending Statements**: Each statement must be concluded by a semicolon. This tells the script to close.
- **Ending Blocks**: A block that begins with an opening curly brace ({) must end with a closing brace (}) following the final statement.
- **Inheritance**: Torque allows you to expand on or override statements within the same script.
- **Variables**: The percent (%) sign before a name signifies that it is a local variable (an invented placeholder currently used within one function). The dollar ($) sign signifies that it is a global variable (a permanent placeholder). You can use alphanumeric characters (A-Z, a-z, 0-9) and the underscore character (_), but you can't start a variable name with a number.
- **Strings**: Constants enclosed in quotation marks are quite often used for in-game messages.
- **Echo**: The `echo()` command prints the value contained in a variable; for example, `echo(''Hello world!'');` will have the words `Hello World!` pop up onscreen.
- **Booleans**: Torque uses Boolean variables, which can have only two values, like true/false or yes/no.
- **Objects**: Definitions of objects come as a collection of attributes and behaviors.
- **Datablocks**: According to the Torque documentation, "Datablocks are special objects that are used to transmit static data from server to client," and they are used for the creation of most objects, from spawning monsters to playing music in the background.
- **Functions**: The basic algorithm you'll see in TorqueScript is a function, which starts, naturally, with the word `function`. If you define a function and then later on define a function with the same name, you'll be overriding the old function completely.
- **Packages**: You can place more than one function into a package, which starts with the word `package` and can be activated with `ActivatePackage()` and deactivated by `DeactivatePackage()`.

- **Classes**: Classes hang the level hierarchy for your game elements, and the core classes include `SimObject`, `SimDataBlock`, `SceneObject`, and `GameBase/GameBaseData`.

Algorithms

An *algorithm* is a set of instructions, listed out step-by-step, to your computer, to make it do what you want it to do. An algorithm is made up of one or more statements.

Your computer thinks in binary, or 1s and 0s, and so you have to slow down and remember to "dumb down" your speech in order to communicate with your computer effectively. You don't literally have to talk in 1s and 0s anymore (thank goodness!), but you *do* have to think logically, critically, and simply.

Imagine that you had to tell a caveman how to operate a car. First, you'd have to explain to the caveman what a car is, and then you'd have to tell the caveman how to recognize a car by sight. Next you'd have to put in plain and specific words, using lots of details he could understand, how to open the car door, get into the car, shut the car door, put the key in the ignition, turn the key in the ignition, and so on. This sounds slow and pedantic, and it is! This is how ponderous programming can seem sometimes.

Human language is full of innuendo and inferences based on shared human experiences that a computer just won't comprehend. When telling a computer what to do, much like telling a caveman what to do, you have to be succinct, direct, and definitive about everything you are trying to communicate.

Algorithms and Human Logic

Algorithms are not the common way we humans think or approach problems. We do not naturally solve challenges in our everyday lives with the use of algorithms. The following will help you understand why:

- We tend to start executing as soon as we start thinking about a problem.
- We tend to tailor our solutions to the problems at hand; in other words, we do not often share standard protocols.
- We rarely think things out in individual steps to take to solve a problem. Minor problems we solve automatically, and others we make automatic inferences about.
- We don't bother to write down our solutions to problems in a step-by-step manner.

- Some things to us seem so obvious because of our shared human experiences, which a computer has no basis for.
- We make minor adjustments to our solutions and change our plan of tactics automatically by observing through a constant evaluation process.
- And lastly, we spend most of our time acting based on emotions and intuition, two things a machine does not have.

A programmer has to come up with an algorithm. An algorithm includes the solution to a problem and the steps by which the machine can solve it. To create an algorithm, a programmer must identify what the problem is, define a solution, turn the solution into an algorithm, program the algorithm, and finally check to see if the algorithm works. It's all very logical once you look at it!

There are many different parts to algorithms that help you out and make programming easier. These are variables, namespaces, data types, objects, operators, functions, and control statements.

Variables

Variables include any observable attributes in a programming language that change their values when ordered to. There are two types of variables in TorqueScript: local and global.

Local variables (`%local_var = value;`) are transitory, meaning they are ruined automatically when they go outside of their scope, or the block of code they are defined in. Let's say you give your player a shotgun and define a "shotgun ammo" variable. This variable would be a local variable, existing for as long as the gamer is playing the shotgun level.

Global variables (`$global_var = value;`), on the other hand, are permanent and subsist throughout the whole program they exist in.

Variable names can contain any number of alpha-numeric characters, as well as the underscore character (_). Do not put a number as the first character in a variable's name, and do not use any wild card symbols. You also do not have to create a variable before it's needed, because when an algorithm attempts to appraise the worth of a variable that has not previously been produced, TorqueScript will construct the variable mechanically.

Data Types

TorqueScript supports multiple data types, including numbers, strings, Booleans, and arrays.

Strings have double quotes around them (`''abcd''`), but strings that appear in single quotes (`'abcd'`) are treated as *tagged strings*, which contain string data but have a special numeric tag associated with them and are important for sending string data across a network.

Arrays used in TorqueScript have a lot in common with the dictionaries of languages like Perl, JavaScript, and Python. In general, you can treat them the same. However, arrays can become confusing very fast and are not suggested for beginners, so we will not delve too deep into their syntax. Just know that array variables can be accessed in multiple ways for a range of reasons.

Objects

In TorqueScript, every item in your game world is considered an *object*, and all game world objects can be accessed and manipulated via TorqueScript. An example of an object found in TorqueScript is as follows:

```
%object = new t2dSceneObject (MyGameObject)
{
      Position = "10 10";
      Size = "5 15"'
      Rotation = 45;
      OtherField = 10;
};
```

In this example, a new object is created, called "`MyGameObject`," and the object is placed at coordinates 10 X and 10 Y.

Operators

Operators are listed for you in the back of this book, in the Appendix A, as there are too many of them to catalog here. If you get baffled and need to know what an operator does, check the Appendix.

Functions

Most statements you'll write in TorqueScript appear as *functions*. These statements follow until you reach the end of a function, and each function builds off descriptions and other functions. This is known as sequential logic.

In the case of telling a caveman how to get into and start a car, you are using a sequential logic. You are telling the caveman, step by step, and you do not expect deviation from the norm. Unfortunately, sequential logic is dull in video games because games are all about randomness and interactivity. If you were simply going to code a CGI movie, you could do it in sequential style, but for games, you are going to have to take into account control statements.

Control Statements

Control statements are statements that involve looping or branching in the program code, offering random arbitration or depth possibilities. They're also good for checking local variables.

If-Else The first popular control statement is the *if-else statement*, which checks against a variable before executing a code. Also called conditional logic, or selective logic, these statements follow the precedents set by British mathematician George Boole (the inventor of Boolean logic). Using conditional statements you can set up whole subroutines to check for interrupts to your sequential logic.

The most popular example of conditional logic used in a video game is the finite state machine. *Finite state machines* (or FSMs) are most prevalent in games as the enemy monsters a player has to beat. The simplest FSM is a monster that is given a set path to wander, such as "go to tree, then to rock, then to barn, and then back to tree." This is a sequential algorithm. It's also a looping one, but we won't go into that right now.

Now let's say that the hero comes across the monster. We want to interrupt the sequential logic by applying selective logic. If the hero is bleeding and unarmed, we want the monster to attack the hero. If the hero is armed and loud, we want the monster to act frightened and run away. We can do this using an if-else statement.

Perhaps the most renowned classic arcade game to use a simple FSM for its core gameplay would be *Pac-Man*, whose joystick many past-generation gamers suckled on in their youth.

In *Pac-Man* (see Figure 5.3), you made your way through a maze while ghosts (Inky, Pinky, Blinky, and Clyde) came out of their crypt to destroy you. Constantly dodging these ghosts while racking up points made for substantial fun. However, if you swallowed the big pill, you became a super Pac-Man and could

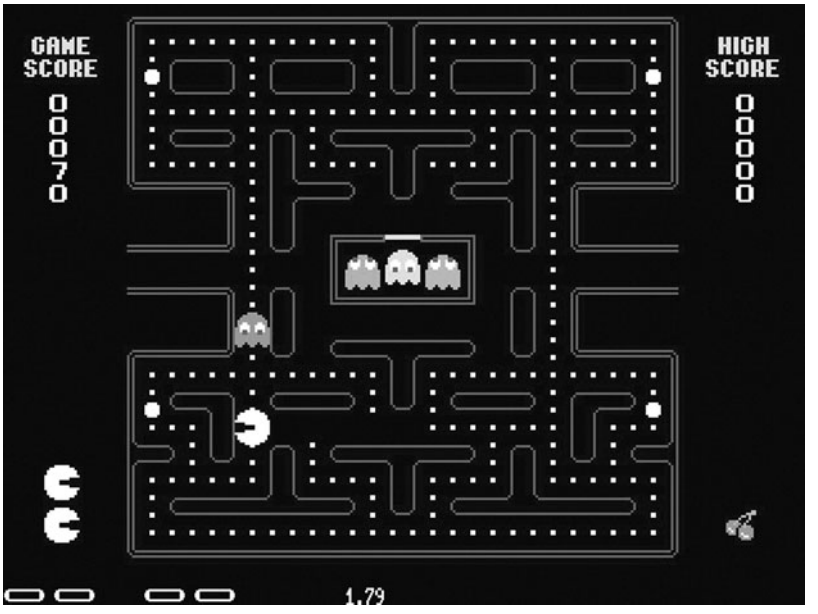


Figure 5.3
An in-game screen capture of the classic arcade game *Pac-Man*, from Namco, shows you the core gameplay where Pac-Man is running for his life from the ghosts Inky, Pinky, Blinky, and Clyde.

eat the ghosts; they all turned tail and ran! The ghosts themselves were programmed as FSMs.

Here's one example of a conditional statement in Torque:

```
if (%val < 0)
{
     %abs = -%val;
}
else
{
     %abs = +%val;
}
```

You could use `if-else` statements for all kinds of game-related items. For instance, you could check to see if the player's health bar is nil, and if so, call a function where his character dies. You could check to see if the player is out of ammo, and if so play an "empty" sound bite instead of gunfire. You could check to see if the player has wandered within the field of vision of a gun turret, and if so have the turret shoot at the player.

Switch Another conditional control statement is the *switch statement*, which checks against a variable and then executes a code based upon context cases. These context cases can be numeric or string related. An example of a string-related switch statement is as follows:

```
switch$ (%val)
{
     case "Doc Brown":
          echo ("Great Scot!");
     case "Marty":
          echo ("Doc, you gotta listen to me!");
     default:
          echo ("What am I doing here?");
}
```

Loops The last control statement is a *loop statement*, which is either expressed as a *for* or *while* statement. All the code within a loop is executed repeatedly, seemingly into perpetuity, until an exit condition is met. The exit condition in a for loop is the second statement in the declaration, while the exit condition for a while loop is between the parentheses after the while keyword. See examples of both, first the for loop statement and then the while loop statement:

```
for (%i = 0; %i < 100; %i++)
{
     echo (%i @ "/" @ 100);
}
%i = 0;

while (%i < 100)
{
     echo (%i @ "/" @ 100);
     %i++;
}
```

Flowcharts

Perhaps the easiest and most visual way for us to conceive of an algorithm is the flowchart method. A *flowchart* is a schematic representation of an algorithm or other step-by-step process, showing each step as a box or symbol and connecting them with arrows to show their progression. With flowcharts, you construct

graphical representations of how an algorithm should work based on the logic that you are using.

Flowcharts are similar to descending trees, as they begin with the start of the algorithm and continue until they reach the logical end, or (in the case of a looping statement) until they reach the `exit` condition. Along the way, different nodes represent sequential steps, selective logic, and loops. Creating these flowcharts is in itself a craft and area of study, which could seriously aid in software development.

The most commonly used nodes represented on flowcharts include:

- **Ellipse**: A circle or ovoid shape usually indicates the start and end of logic, as well as events along the path.
- **Square**: A square is indicative of a sequential step in the algorithm. Sometimes developers will use a square for an inactive step and a parallelogram for a function statement.
- **Diamond**: A diamond shows the viewer where selective logic comes into play, and it often has two (or more) arrows leaving it, depending on the current status returned.

See Figure 5.4 for an example. This covers checking to see what's wrong with a fridge, if it's working correctly. Most of the steps are obvious, but remember when dealing with a computer you have to go beyond the obvious and be very detailed in each and every step along a routine.

Torsion

Tip

> When I tried Torsion, I was able to set and hit a breakpoint in just a few minutes. My advice is to not waste your time with any of the others.
>
> —Jamie Fristrom

Created by dedicated Torque developers at Sickhead Games, Torsion maximizes your overall output when working on your project based on the Torque Game Builder. Torsion is a syntax editor used for programming your games, but unlike other syntax editors, Torsion solely targets TorqueScript to ensure a focused tool without features for other engines getting in the way.

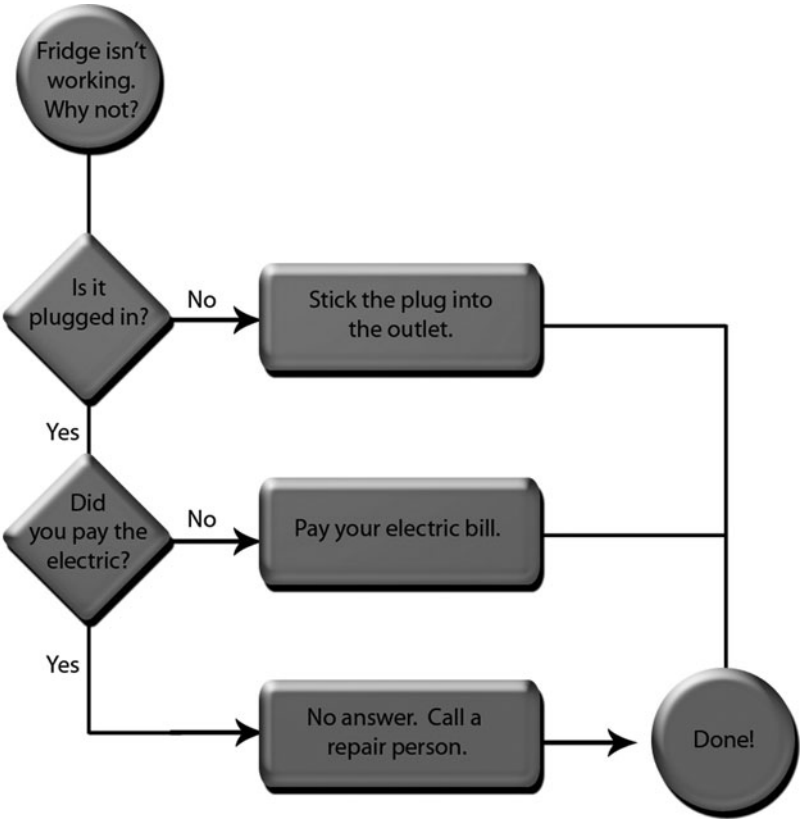


Figure 5.4
A simple flowchart depicting what to do to check to see what's wrong with a fridge—why it's not working correctly.

A *syntax editor*, as opposed to developing in a simple text editor, highlights and color-codes your program code to make reading the code easier for the programmer. Users familiar with other modern development environments will feel right at home using Torsion.

Torsion contains everything you would expect in a modern IDE:

- A familiar and intuitive user interface.
- Advanced editor features like code folding, line wrapping, auto-indent, column marker, automatic bracket matching, and visible display of tabs and spaces.
- Customizable syntax highlighting for TorqueScript.

- Unlimited undo/redo buffer.
- Goto line and text searching.
- ScriptSense updated dynamically as you type.
- Smart project tree view allows for file manipulation and automatically updates as new files are added.
- Code browser window for exploring both engine exports and script symbols in your project.
- Integrated "One Click" script debugging.
- Full control over script execution via `step` and `break` commands.
- Conditional breakpoints.
- Call stack and watch windows allow you to fully inspect and change the running game state.
- Remote console for executing commands in your running game.
- Complete editor state saved between project sessions including open files, breakpoints, and variable watches.
- Can be easily copied and run from a flash drive.
- Written entirely in C++, making it fast and efficient with a small memory footprint.

There is a demo copy of Torsion found under Software Demos on the companion CD to this book, or you can go to Sickhead Games' website at http://www.sickhead.com to find the latest download.

Other Syntax Editors

If you decide you like using a syntax editor rather than a text editor for coding purposes, then you might try some on this list. The following are recommended syntax editors for use with Torque:

- **Torsion** (http://www.sickheadgames.com)
- **TorqueDev**, a.k.a. **CodeWeaver** (http://tdn.garagegames.com/wiki/TorqueScript/IDE/Guide/TorqueDev)
- **jEdit** (http://www.jedit.org) with **TIDE** (http://torqueide.sourceforge.net)

- **Tribal IDE** (http://www.garagegames.com/index.php?sec=mg&mod=resource&page=view&qid=1413)

The following are other syntax editors not primarily intended for TorqueScript but which can be useful:

- **Brain Editor Professional** (http://www.twinno.com/brainedpro)
- **ConTEXT** (http://www.context.cx)
- **Crimson Editor** (http://www.crimsoneditor.com)
- **Eclipse** (http://www.eclipse.org) with **TorqueEdit** (http://opensource.kruxgames.com/torquedit)
- **EditPlus** (http://www.editplus.com)
- **Emacs** (http://www.emacs.org)
- **Jen's File Editor** (http://home.t-online.de/home/Jens.Altmann/jfe_eng.htm)
- **Lemmy** syntax (http://www.softwareonline.org/products.html) plug-in for TorqueScript (http://www.skippingrock.com/torque)
- **MultiEdit** (http://www.multiedit.com)
- **SlickEdit** (http://www.slickedit.com)
- **SourceEdit** (http://sourcedit.com)
- **UltraEdit** (http://www.ultraedit.com)
- **Vim** (http://vim.sf.net)

Just be sure to pick a scripting environment you feel comfortable in. It's more important to work efficiently and in a timely manner without constantly fighting with the interface.

Review

After reading this chapter, you should understand the following:

- What a programming language is and how it can be used to make games
- The syntax used for the TorqueScript programming language
- How to set up basic algorithms, including finite state machines (FSM)
- How to design a programming flowchart
- What Torsion is and how it could benefit your development

This page intentionally left blank

CHAPTER 6

Your First Game Project

In this chapter, you will learn:

- To adjust your camera view and design resolution of your workspace
- To import resources into your game project
- One method for level building in the TGB Level Builder
- To create animated characters for your game
- To play through and test your game

You've looked at general game design, the secrets behind 2D game graphics, and programming games using TorqueScript. You are ready to begin using Torque Game Builder for real, now! In this chapter, you'll devise an approach for setting up your personal workspace and begin your very first game project using Torque Game Builder.

Getting Started with TGB

If you haven't already, it's time for you to install the Torque Game Builder demo. A demo is available on the CD-ROM disc that ships with this book, so you can install it from the CD, or you can go to the GarageGames website at http://www.garagegames.com and look under the Products drop-down menu for Torque Game Builder and the Download TGB Demo button (as seen in Figure 6.1).

Figure 6.1
The Download TGB Demo button on the GarageGames website.

Choose the demo download appropriate for your system, be it Windows, Macintosh, or Linux. Run the .exe file and follow the onscreen prompts. Note that the TGB demo on the companion CD, and the one used throughout the following exercises, is version 1.74. If you have a different version, some of the instructions that follow may not be the same.

When you are through installing TGB, launch the program. Once greeted with the welcome screen, you should see two big buttons in the very center. One says Open and one says Create. Click on Open for now and browse to the GarageGames\TorqueGameBuilder-1.7.4\Games\BehaviorShooter folder installed on your computer. Select the project.t2dProj file and open it. You will see a spaceship on the left and a desert scene.

Adjusting Your Workspace

When creating any new project in Torque Game Builder, it is vital to know how to adjust your workspace to improve your personal workflow. The following tips will show you how to change your camera view and set your design resolution. How you determine to use either is up to you, because you're the designer and you should know how you want your game to be viewed.

Your *camera view* (in the scene window) is set by Torque Game Builder as a box with a blue border surrounding it. All that is visible to the game's player is what is inside of this square. Let's place something inside our scene so that our camera view can see it. Click and drag any of the images available in the Static Sprites library under the Create tab on the right-hand side of your screen into the center of your scene on the left.

When an image is placed in the current scene, it is automatically resized to fit the design resolution setting. The *design resolution* is the size that Torque Game Builder assumes your camera view is. By default, the design resolution is set to 800×600, and therefore any graphic dragged into the scene will be resized relative to 800×600.

Camera View

The Camera tool allows you to manually change the size of the camera view. Its icon is located on the top bar of the Level Builder, as seen in Figure 6.2. When you click the Camera Tool icon, your view zooms out to show you your camera view as a sheer box with resize handles along the outer edges.

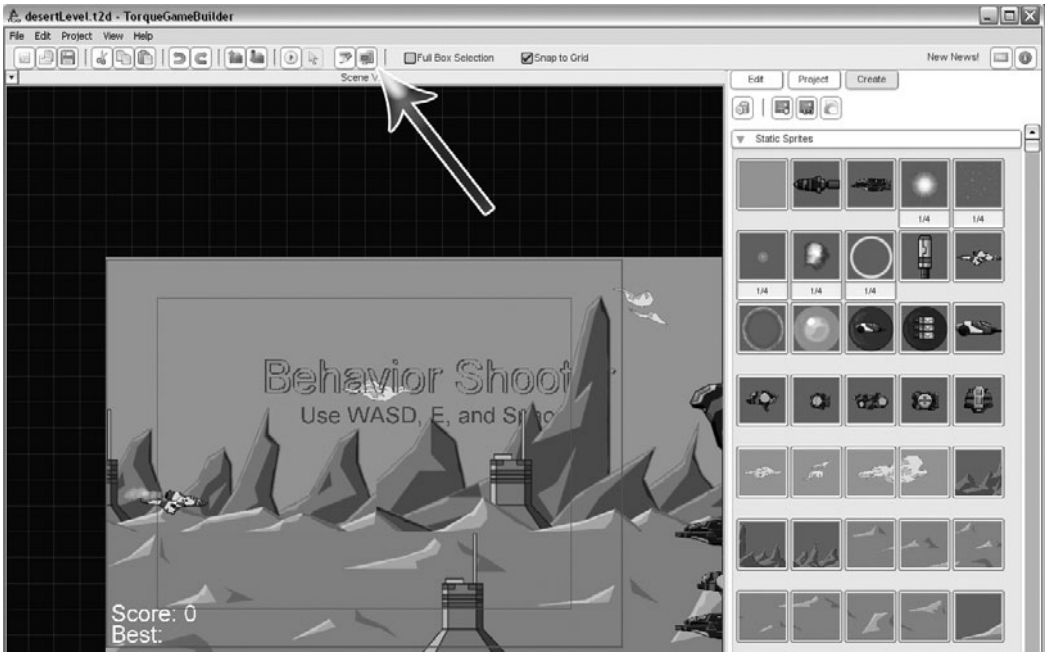


Figure 6.2
The Camera Tool icon is found on the top bar of the Level Builder.

You can resize the camera view in the following ways:

- **Click and drag the resize handles in any direction**: the camera view will resize around its origin and keep its default aspect ratio.
- **Hold down Ctrl and click and drag the resize handles**: the camera view will resize with an anchor in the opposite corner as the one you're dragging from.
- **Hold down Shift and click and drag the resize handles**: the camera view will resize around the origin but will *not* keep the aspect ratio (definitely not recommended!).
- **Hold down Shift + Ctrl and click and drag the resize handles**: you can change the camera view freely, altering origin and ratio (again, not recommended!).

To save the changes made to your camera view, press Enter or click on the selection tool icon.

Design Resolution

As mentioned above, design resolution sets the size of the camera view and therefore the relative resizing of graphics placed on-screen. To access your design resolution settings, go to Edit > Preferences and click on the Scene Editor tab in the Options dialog window that comes up, shown in Figure 6.3. About halfway down, on the left-hand side of the Level Editor Options window, you should see Design Resolution, which defaults to 800 × 600.

Try changing the Design Resolution values to 1280 × 1024 to see the effect it has on your graphics. Click the OK button after inputting the new values, then click and drag the same image you chose before from the Static Sprites library under the Create tab on the right-hand side of your screen into the center of your scene on the left, next to the original one you'd placed. Notice a size difference? This is because the design resolution setting has been changed. Once you're done playing around, adjust your design resolution back to 800 × 600 and close TGB (without saving).

Creating a New Project

Creating a new project in Torque Game Builder couldn't be simpler.

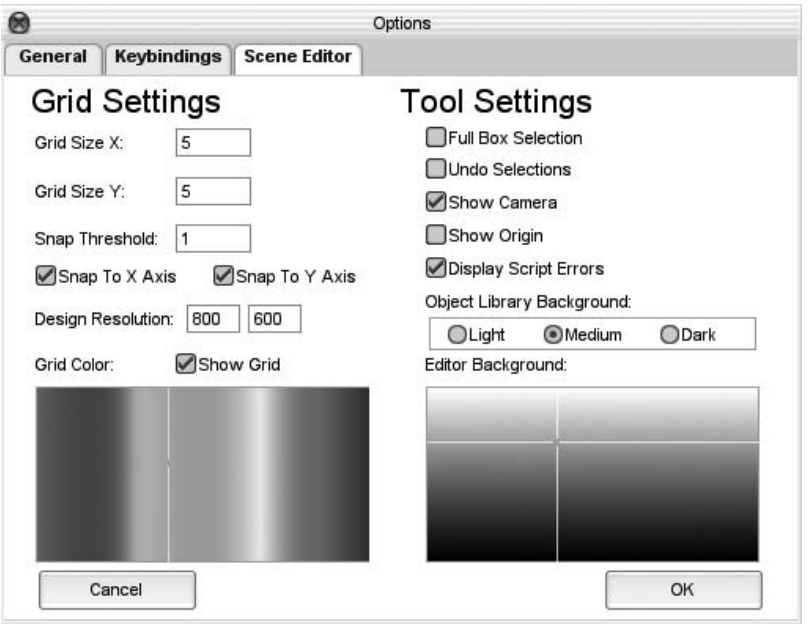


Figure 6.3
The Design Resolution shows up in the Level Editor Options window.

1. With the Level Builder window open (after launching the program), go to File > New Project on the main menu.

2. You will be presented with a dialog box asking for the name of your new project. Always name your projects something simple and descriptive. For the purposes of the following exercise, call this new project "MyCastle" (see Figure 6.4).

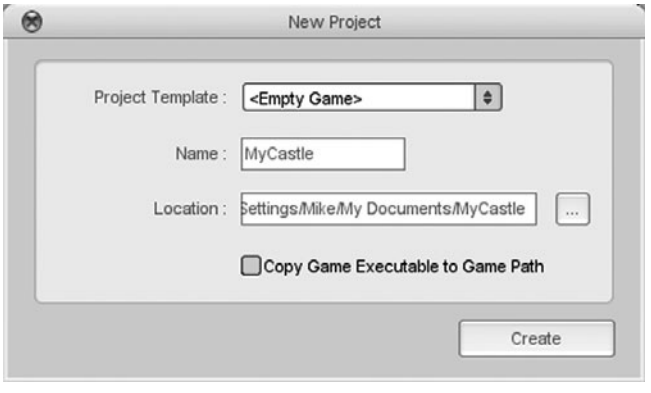


Figure 6.4
Call your new project "MyCastle."

3. Unless you are conducting the Fish Demo Tutorial from the sample tutorials that ship with Torque Game Builder, you will most likely choose "Empty Game" from the Project Template. This creates an empty game project, without any template objects.

4. Click on Create when you are done.

Saving Your New Level

As soon as you have created a new project, you are presented with a blank level, by default called `Untitled.t2d`. To save your level as something different:

1. Go to File > Save on the main menu. The file browser dialog window will open.

2. Browse to your MyCastle\Game\Data\Levels folder, because this is where you will put your game's levels. This should show up by default, so you might not have to browse at all.

3. You will want to save your game levels as something meaningful so that you'll remember which one's which. For the purposes of this exercise, type "MyCastleLevel" for the filename.

4. Click the Save File button.

From now on, to save changes to your current level as you make them, you can click the Save icon to instantly save your work. Remember to save often, as you don't want to waste a few hours' work when your computer decides to leave you in the lurch and you realize, forsakenly, that you had forgotten to save!

Loading Your Resources

For the purposes of this exercise, there are downloadable data files that come with this book. After downloading them and installing them on your hard drive, they will be available for your use. Before continuing with this exercise, you will need to copy the files you need from these data files and place them into your work folder.

1. Momentarily leaving the Torque Game Builder Level Editor, go to the Chapter 6 folder in the data files found on the CD-ROM.

2. Select all the image files inside the Chapter 6 folder and Edit > Copy (shortcut Ctrl + C for Win users, CMD + C for Mac users) to copy them to your Clipboard.

3. Browse to your MyCastle\Game\Data\Images folder and paste the image files from your Clipboard into it by using Edit > Paste (shortcut Ctrl + V for Win users, CMD + V for Mac users).

That's it! You are ready to build your first game level in Torque Game Builder.

Level Design

Without an appropriate frame of reference, evidenced by a game world, the player is just a random piece without a place to be. Gameplay has to have a background for a frame of reference, a playground for the player to explore, find resources, and beat combatants in. If we use the analogy of the theater, the game world provides the stage for the action, and players are the actors on the stage. Inside this game world there are several scenes, or chapters, we call levels. Each level is supposed to have an objective for the player to reach before they can continue on to the next. *Level design* is formally defined as the creation of environments, scenes, scenarios, or missions in an electronic game world.

Tip

"Most often, when you think of game art, it is the characters and creatures that come to mind—even though it is the environments those creatures inhabit that really create the illusion of a world. . . You want your game to be convincingly real to truly inspire mood and drama—and yet you have to be inventive without straining credulity. It's a matter of combining and synthesizing, keeping the aspirations in your head but looking for new ways to fit it all together."

—Marc Taro Holmes, Obsidian Entertainment

Level Structure

Levels are basic units, like chapters in a book, used for subdividing and organizing progress through a game. When designing game levels, you should take into account their goal, duration, scale, and difficulty.

Goals Every level must have at least one goal that is spelled out clearly and concisely to the player. It can be a simple goal, as you see in the *Donkey Kong Country* games on the N64, to get from the left side of the screen to the right side of the screen without perishing in the process. Some games have short

"briefings" built into the start of each level, either as a text document to be read or a short cutscene that ensures players know the coming objectives. Other games thrust the player right into the middle of the action and engage them in tasks where the player must infer the goal. If you build very long or complex levels, you might consider attaching a goal tracker of some kind, even if it must take the form of progress reports, to let the player know how close they are to achieving their goal so they don't give up hope.

Duration You must take into account when building a level how long you expect the player to spend in it. You first want to make sure that a player stays in a particular area long enough to accomplish necessary objectives, but there's a nearly universal rule that a player must be able to complete at least a single level of your game within one session. This can be anywhere from 15 minutes to 45 minutes, depending on your audience, the complexity of the game, and your intentions with the game. The longer levels should have milestones or goal trackers placed in them so players don't get frustrated.

Scale The scale of a game space includes the total size of the physical space and relative sizes of the objects in the game as juxtaposed to the player's avatar. Since most simulation games try emulating reality, the space and objects are scaled to relative size. In most games items like keys, weapons, and other important objects are exaggerated so that the player can easily spot them. Scale can be distorted to demonstrate to the player an object's importance as well as to make an artistic statement.

Scale in games is an illusion, and therefore even seemingly realistic environments are unrealistic distortions of reality. For example, look at a game like *Grand Theft Auto: San Andreas*. You can navigate across an entire city in the same amount of time it takes you to have a short firefight in a hotel building interior.

Difficulty Early levels in the same game should be less difficult than later levels, with some exceptions. This is called *ramping*, which means that a game gets increasingly harder the longer someone plays it. The theory behind ramping is that players will get superior over time with continued exposure to the same play system. Regular players will even develop their own subsystem of shortcuts for beating the game quicker. So ramping gives them a better challenge throughout their game experience.

However, every player needs a break now and again, so even games that use ramping will throw in easier "breather" levels to let the player "catch their breath"—especially after boss fights or really challenging levels!

Tip

"Level design is the process of taking the world envisioned by the lead designer and the design document and creating a world filled with goals, subgoals, and lots of obstacles between the player and those goals. The level designer is responsible for designing the level's world; determining which enemies, weapons, and power-ups appear in the level; how and when the enemies appear or spawn; and the overall balance of the level (so the level is neither too easy nor too hard)."

—Richard Wainess, Senior Lecturer at the University of Southern California

Gimmicks

A lot of games, starting in the early years of SEGA and Nintendo onward, contain gimmicks. *Gimmicks* are archetypal level types that are immediately memorable for players because of their familiar themes. Gimmicks can be so overused to the point of becoming cliché, but gimmicks also give character to dull or unoriginal game levels when used appropriately, which is why they are still used with regularity.

The most popular used gimmicks include:

- A sewer level, full of noxious fluids to wade through (see Figure 6.5)
- A lava level, where one wrong step hurts

Figure 6.5
You can combine many gimmick levels in one, including this noxious toxic sewage area that is just as dangerous as any lava level.

- A graveyard level, where the dead come back to life
- A mine cart ride, where gaps in the rails can be fatal
- An underwater level, where the player's character swims with the fishes
- A snow level, where surfaces can be slippery

Try making your levels fresh rather than parodies of clichéd games by putting a new spin on gimmicks. For instance, you can achieve the same fun found in snow levels by placing your game in an ice cream factory overflowing with the frozen desert! Put more thought into your levels than simply copying game levels you've seen before.

Art Style

Tip

> "Even though it's fun to crawl inside a computer and play with its potential, it's really important to look at other aspects of your life as well. That's where ideas for programs will come from."
>
> —Marcia Burrows, K-Power Magazine

Most level designers use the real world as inspiration. Many of them actually go location scouting and take multiple photographs or make sketches of unusual and interesting objects, buildings, textures, and environments. They incorporate this research into the worlds they build for games.

The level designer's own personal artistic style will carry throughout the level and create a rhythm and mood that pulls it all together. Artistic style influences everything, including the way characters look, the way objects look, and the way backgrounds are perceived. Look at Tim Schafer's game *Psychonauts*, which has one of the wildest levels ever designed in a game: the Milkman Level. The Milkman Level recreates an M.C. Escher-like art piece and a Moebius strip inside a game (seen in Figure 6.6), and is unique because the laws of physics no longer apply.

Cultures can affect the art style of a game, as well. Dusty Western shooter games are affected by the American pioneer days of the Wild West. Games like *Voodoo Vince* and *Shadow Man* spin a romanticized appearance of New Orleans and its jazz and voodoo atmosphere to evince a cultural style (see Figure 6.7). The same can be said of made-up cultures in fantasy role-playing games. Each culture can be more distinct than the next, and the application of cultural style can change the look and feel of one game level to separate it from the next.

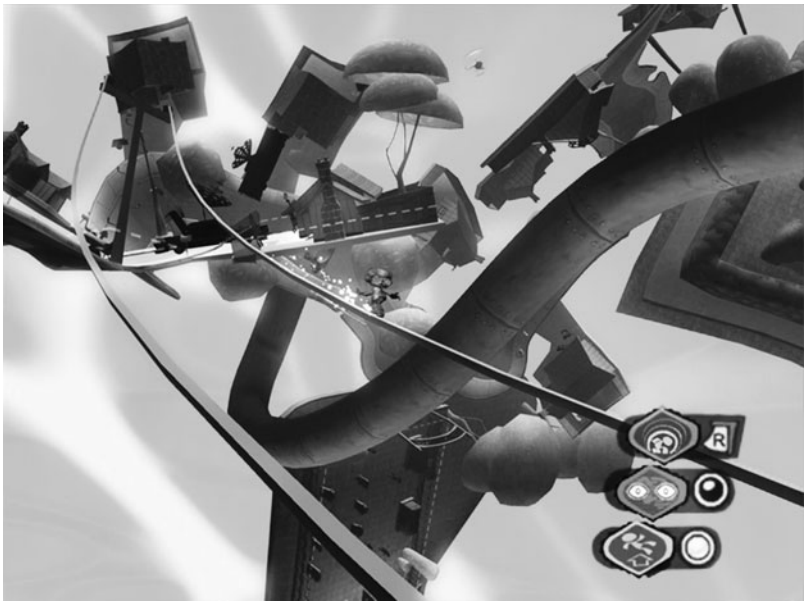


Figure 6.6
Games are cool, but walking on walls decidedly takes the cake!

Figure 6.7
A game can capture the jazz and luster of old New Orleans while paying homage to her voodoo roots.

One of my rules of thumb when developing game levels is to consider the game world as a giant amusement park or vacation resort and your player is your tourist. People love to play games, watch movies, and read books about far-off locales, pretty landscapes, and places they themselves wish they could take a trip to. It's a basic form of escapism. Get it in your head that, as game designer, you are delivering a whole package: treat your players as possible tourists and set them up for thrills in lush and exciting game worlds!

Tip

"Reality is merely an illusion, albeit a persistent one."

—Albert Einstein

Game Flow

When originally creating a game outline, your game can be long or short, depending on the built-in challenges, obstacles, and resources. It also varies based on the detail of the architecture. Simply put, a game where each level is supposed to be the size of Las Vegas will be very different from a game where each level is supposed to the be the size of your parents' house.

The pacing of game action, what is called *game flow* by industry professionals, can be short and sweet or long and meditative and is often dictated by the size and complexity of the game's levels and their content.

Some game developers actually prescribe to the belief in *feng shui*, or the theory that people's moods and reactions can be orchestrated by the placement of objects in a room or (in this case) on the computer screen. Learn to pace your game's flow by proper positioning of entities in your game level as well as the architecture of the level itself.

Industry Insiders Dos and Don'ts

From leading industry professionals today, here's a compiled list of the top tips in design to keep in mind.

- Don't forget to give the player enough help so he can survive.
- Don't design parts of a level that are so difficult they form a "choke point."
- Don't design levels so confusing the player has to read an online walkthrough to finish.
- Don't forget to accommodate new and experienced players.

- Don't reveal all your best eye candy in the first level.
- Don't keep fiddling with a level; design it and move on. You can always return and make edits later.
- Don't forget to test your level as you go.
- Don't frustrate or make your player mad at you.
- Avoid areas that look important but really aren't.
- Don't put all your monsters or power-ups in one secular area. Proper placement is important.

Making a Level in TGB

You will use the TGB Level Builder to create game environments. What follows is one method for creating a standard 2D game environment.

Adding and Scaling a Background Image

One of the resource files we brought into our Data\Images folder is a castle background image we can place into our level.

1. Click the Create tab on the right-hand side of the editor.
2. Click the Create a New ImageMap button under the tab.
3. In the dialog box (shown in Figure 6.8) you can locate the castle.png file.
4. To open the castle, click the OK button. The castle will become a sprite in the Static Sprites section of your editor, under the Create rollout (seen in Figure 6.9).
5. Under the Create tab, go to the Static Sprites rollout and drag-and-drop your castle background image into your level on the left.
6. Notice after dragging the castle background image into your level that it is too big. This is because the image was specifically created large enough to scale properly with end users' computers at higher resolutions. So you will have to scale it down to make it fit properly.
7. Zoom out by scrolling with the middle mouse wheel or the zoom out button.

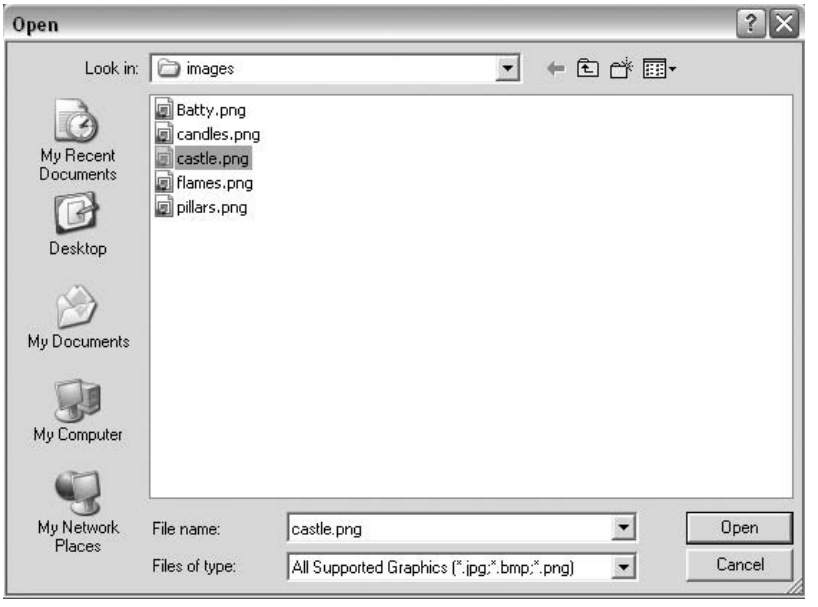


Figure 6.8
Finding the MyCastle\Data\Images folder in the Current Path section.

8. Grab a corner handle of the background image. Hold down the Shift key and drag the corner handle. Remember: holding down the Shift key while clicking-and-dragging corner handles resizes but keeps the image's overall ratio the same.

9. Your purpose with the castle background image is to make it just slightly larger than the camera view, or the thin border line behind your image. To do this, simply scale the image to fit just barely within the camera, and then scale it out so that its edges rest just beyond the camera view (compare your work to Figure 6.10).

Layering the Background Image

There are two more background elements we need to add. Before we do that, however, we have to set the main background image's layer. Remember, there are 32 layers and the lower the numbers, the closer the layer is to the camera, so whatever you place on furthest layer, layer 31, will be behind every other object on the screen, and whatever you place on layer 0 will be right up next to the camera, or in front of all the other objects on the screen. Keep in mind that every new image you add will have a default layer setting at 0.

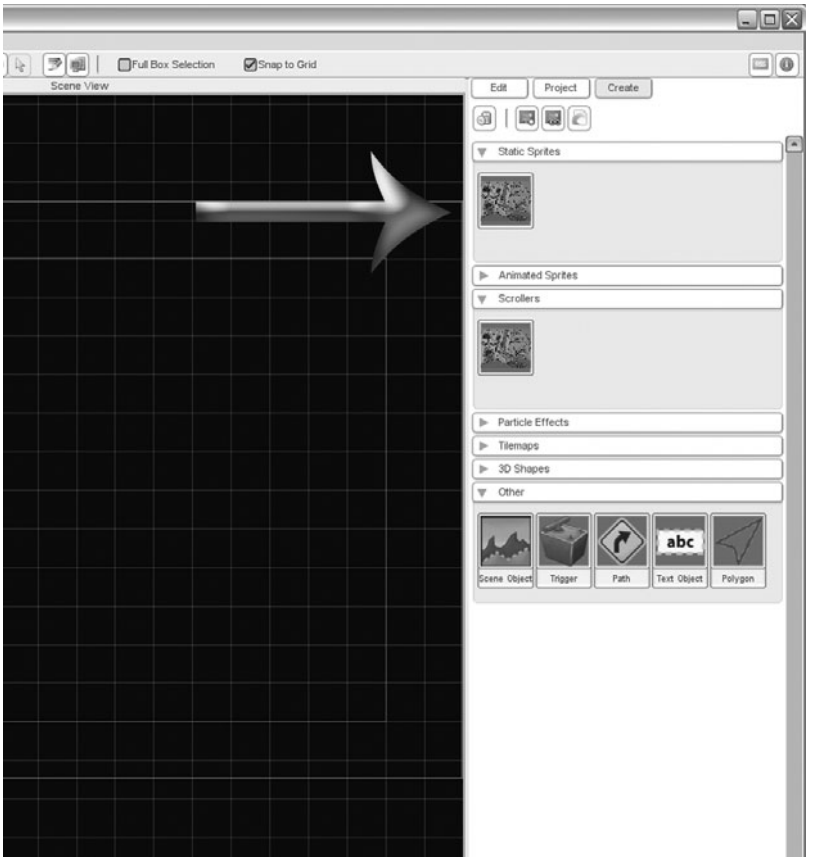


Figure 6.9
After saving your new image, you should see it appear in the Static Sprites section of the Create rollout tab.

1. Make sure the background image is selected.
2. Click on the Edit tab and scroll down until you find the `Scene Object` property.
3. Since this is our background image we want to set it farther back than most of the other layers, so set its Layer to 30 (see Figure 6.11). Enter the value 30 and press Enter, or press the right arrow 30 times until you reach the desired value. The reason you want to leave a layer or two open back behind this one is because you never know when you might want to add another element that has to move out of view entirely.

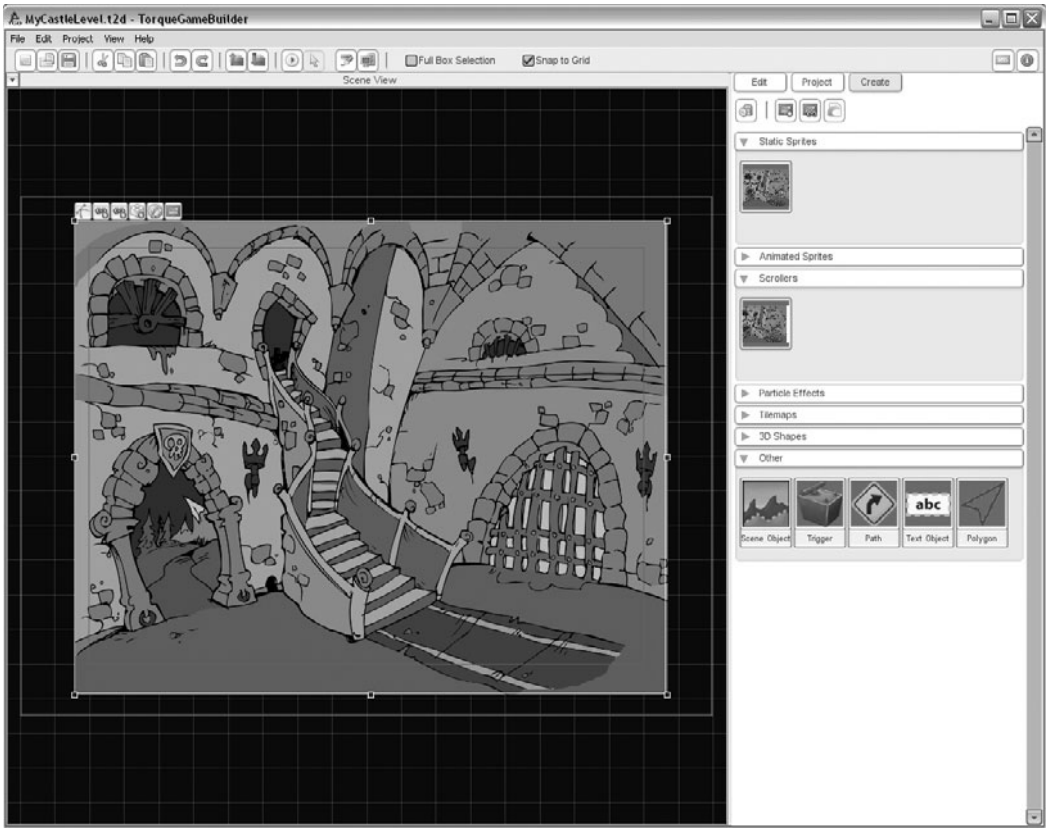


Figure 6.10
Adjust your background image to fit the camera view.

Adding and Layering Other Background Elements

Now that you've added, scaled, positioned, and layered your full screen background image, it's time to repeat the process with our other background elements: our candelabra and our pillars. We will want them to be on a layer between our castle background and our characters, but it's a smart idea to leave a bit of virtual gap between the layer images just in case we have a character walk between them.

1. Go to the Create tab on the right-hand side of the editor.
2. Click the Create a new Image Map button under the tab.
3. Find the file candles.png.
4. To open the candles, click OK. The candles will become a sprite in the Static Sprites section of your editor, under the Create tab.

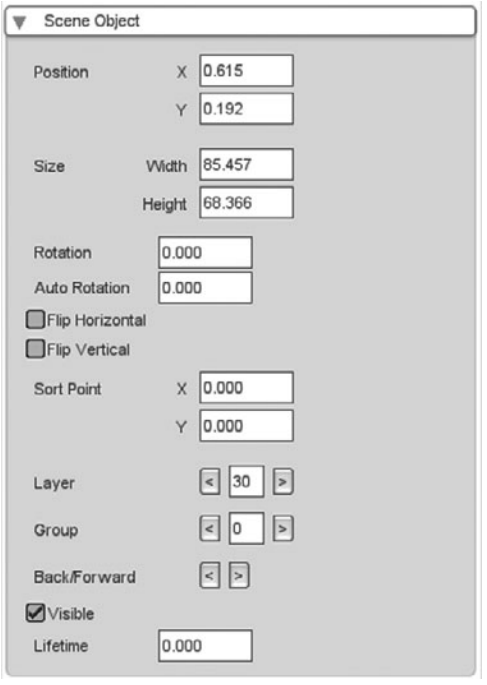


Figure 6.11
Set the Layer object property of your background to 30.

5. Under the Create tab, go to the Static Sprites section and drag-and-drop your candelabra image into the center of your level.
6. Just like you had to do for our background image, hold down the Shift key and click-and-drag one of the corner handles of the candelabra image to resize it proportionally with the environment. Compare your work to Figure 6.12.
7. With your candelabra still selected, click the Edit tab, scroll down to the Layer object property, and set the value at 20.
8. Go back to the Create tab.
9. Click the Create a New Image Map button under the tab.
10. Find the pillars.png file.
11. To open the pillars, click OK. The pillars will become a sprite in the Static Sprites section of your editor, under the Create tab.

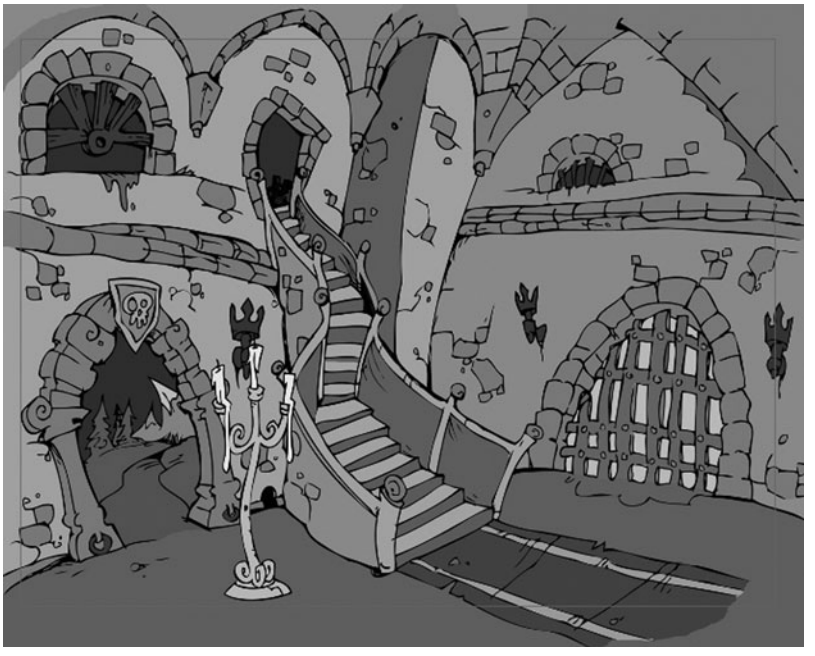


Figure 6.12
Set up the candelabra in your castle scene.

12. Under the Create tab, go to Static Sprites and drag-and-drop your pillars image into your level and place them at the base of the staircase on the right, somewhere where they'll look good.

13. Scale the pillars to fit proportionally with the background image. Compare your work to Figure 6.13.

14. With your pillars still selected, click the Edit tab, scroll down to the Layer object property, and set their value at 6.

There's just one more element missing! Our candles on our candelabra should be lit! To attach this, there's a simple animated sprite of candle flames that we can bring in and insert into our scene to complete it.

1. Go to the Create tab on the right-hand side of the editor.

2. Click the Create a New Image Map button under the tab.

3. Find the flames.png file. Click OK to move them to your Static Sprites section.

4. Under the Static Sprites rollout, find your flames and double-click on their thumbnail to open the Image Builder dialog window. Here I want you to

Figure 6.13
Set up the pillars in your castle scene.

change the Image Mode to CELL (see Figure 6.14). Cell mode is used for images that need to be evenly divided into more than one frame. The default Cell setting is four equally-sized frames. This is perfect for flames, so we don't have to change a thing.

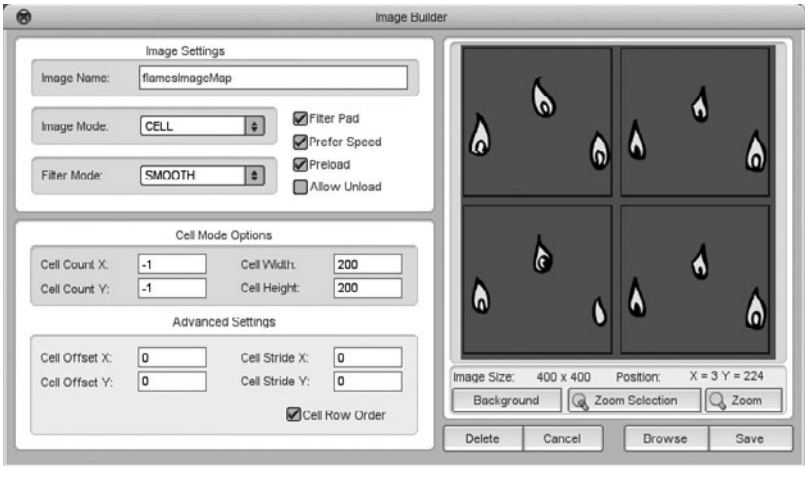


Figure 6.14
Change the Image Mode of the flames image to Cell.

5. Click Save and `flamesImageMap` and notice that its caption in the Static Sprites section has changed to say "1/4". This is because frame one of four is currently showing.

6. Under the Create tab, click on Create a new Animation and the Select Material dialog window will open.

7. Choose the `flamesImageMap` as the source image (it can be found on the left, under Source).

8. Click the green plus sign button that says "Use all frames from image" when you hover over it and then click the Play Animation preview button on the upper right to see how the candle flames will look animated. Set the Frames Per Second field to around 9, to slow the animation down, and also check the Random box. Compare your work to Figure 6.15.

9. Click Save and your animation will now appear under the Animated Sprites rollout.

10. To place the sprite into your game, simply drag the sprite image into your level and resize it to fit your candelabra image. Just like you did before, you can hold down Shift to keep resizing the image proportionally when dragging on a corner handle. Compare your work to Figure 6.16.

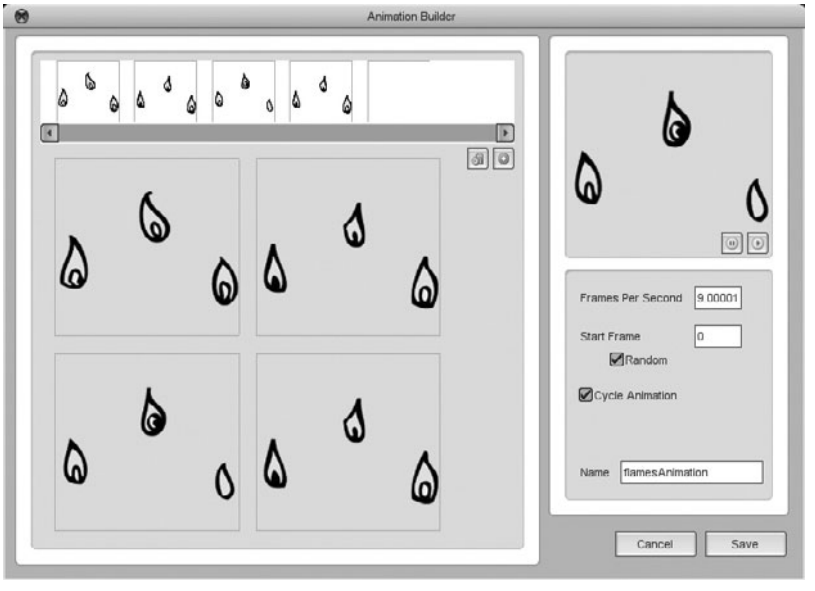


Figure 6.15
Preview the animation of your flames.

Figure 6.16
Set up the animated flames in your castle scene. Trust me, these boys are dancing!

11. Once you have your animated sprite image to the size and position you desire, and it looks good, you can then click on the Edit tab on the right-hand sidebar and therein change the object's Layer property to 19. This puts the candle flames above the candelabra's layer but behind the pillars' layer.

You have now placed all your background elements and composed a fairly decent level. Since your base level is complete, you are ready to add your character!

Designing Characters

Tip

"Our character is what we do when we think no one is looking."

—H. Jackson Brown, Jr.

Great characters, such as Lara Croft, Mario, Cloud, Agent 47, Kirby, and more, transcend the visual screen into iconic figures of the games they serve. Their names and appearance evoke whole feelings and memories in gamers, because they are the imaginary people whose eyes the players have looked through. They become truly memorable characters in their own right. Sometimes a memorable character appears to become a legend overnight, but indeed it is the struggle and vision of the designers behind the characters that lead to their ultimate success.

No character has ever started off fully-formed but grew out of careful design and planning.

First, designers have a powwow and brainstorm the character. Then character sketches and color mock-ups are drawn up. Amid countless variations, a design is chosen and launched into the 2D or 3D computer counterpart. After further edits, the image is refined. Finally, when the game starts and the player takes control of the character, it's the player's decision-making skills and actions in-game that define the character for who it is.

Many designers today say it's not important to have a strong central figure in a game (also called an *avatar*), that the player's own personality and choices should truly take center stage. If the game's main avatar is given too much of a life all its own, it might rob the spotlight, they say. Flat nondescript characters often result from this line of thinking, but you never see games win popularity contests with these sorts of amorphous characters. Even fantasy role-playing games that have an ambiguous ''you'' as the central character don't compare to the ratings you see come from games like the *Final Fantasy* series, which has plenty of idiosyncratic characters that become rock stars in their own right (see Figure 6.17).

Figure 6.17
Fantasy RPGs are well-known for their idiosyncratic characters.

Players like to watch Harry Potter and Indiana Jones and wonder what it might be like to be them. This sort of open make-believe governs the hub of the most popular games on the market today.

I'm not suggesting you rip-off popular media characters. You should always use your imagination and come up with original characters that appear fresh, cool, and fun to play. Keep in mind that they should be exciting and never flat or unappealing. To help you decide, here are some of the common attributes of memorable game characters:

- Name
- Personality
- Appearance

Note

The word *avatar* comes from ancient Sanskrit and means "the incarnation of a Hindu deity in human or animal form." One of the most popular examples of this incarnation would be Buddha, considered to be an avatar of Vishnu. In this sense, an avatar in a game is an extension, or incarnation, of the player. It's no wonder players identify so strongly with their avatars!

Name

So far there's only been one nameless character, Nameless One from Black Isle Studios' *Planescape: Torment*, which has been successful. It is generally a practice discouraged, for very good reasons (see Figure 6.18). Your character's name reflects the character, the player, and the company that made the character, and should not be chosen lightly.

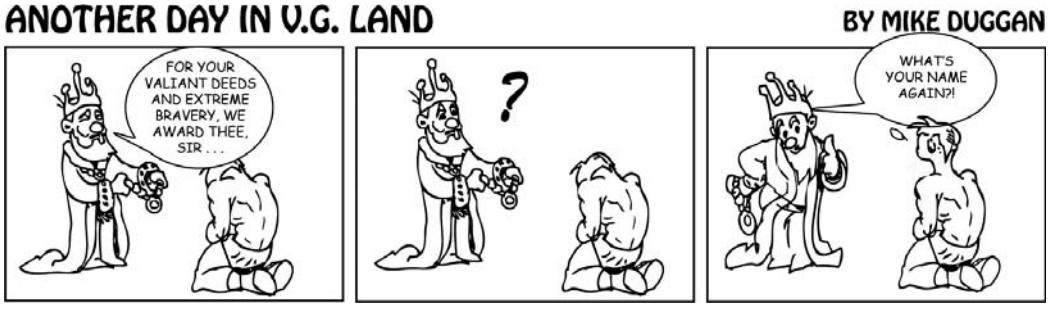


Figure 6.18
It's worked once before, but don't name your character Nameless One!

Names influence the way people act and react as the character. Compare the name Duke Nukem to the name Guybrush Threepwood. The former sounds like a tough macho guy, but the latter sounds like a brainiac or simp. A character's name should fit the character and should also fit within the game's environment. It should also be short enough the player can become instantly familiar with it, and thereby a household name.

Also, avoid making names that are too similar to one another or homonyms. You run the real risk of confusing the player or causing her undo frustration if she has to remember some character based solely on his or her name. Having characters named Chris, Kristin, and Krys all in the same game is a huge no-no! The player will have a hard time telling them apart.

One of the best tricks to coming up with names is using a baby name book, which you can find free in your local library. Baby name books also describe the origin of names and their meanings, which might give your imagination a kick-start.

Personality

Creating a memorable character players want to play over and over is definitely rewarding, from a commercial end as well as an artistic one. One of the oft-repeated critiques of *Mario 64* was that players like being Mario, even when they weren't actually doing anything with him. It was just fun to run around, jump, and double-jump, listening to his funny hoots of excitement.

Giving the game's main avatar likeable personality traits will cinch your player's interest. Personality traits can not only influence the player's choices, they can also shape the character's looks, actions, and dialogue. Before you finish designing your character, make sure your character has at least three remarkable personality traits. Some designers actually take the "old school" approach to this and fill out character sheets full of information about each and every character they put into a game, much like the paraphernalia used for tabletop role-playing games such as *Dungeons and Dragons*.

Here are some examples of popular media characters and their personality traits:

- **Shrek** (from *Shrek*): ugly, heroic, and clumsy
- **Donkey** (from *Shrek*): smart-mouthed, energetic, and conscientious
- **Ash** (from *Pokemon*): overconfident, willful, and eager
- **Spock** (from *Star Trek*): intelligent, logical, and repressed

Biographer John E. Mack wrote in *A Prince of Our Disorder*, "A vital ingredient in hero-making is the resonance that the follower finds between the conflicts and aspirations of his own and those he perceives in the person he chooses to idealize. . ." In other words, find problems we all share as individuals and make the player fantasize through the main avatar about the ways we can actually solve them, real or not. *Rolling Stone* magazine reported that we live in the age of the slacker-hero for this very reason: people like to root for the underdog, because the underdog reflects that part in all of us. This is one of the key ingredients that led to the success of the *Grand Theft Auto* game series.

Appearance

So far, every detail I've mentioned about creating memorable characters for games applies equally as well for fiction as it does games. However, games are by essence a visual media, and so every character you design for a game has to make an appearance. Designers find the greatest challenge to making a character is evincing a strong reflection of the character's name and personality in a single visage that can be glorified in 2D or 3D.

Game characters are expected to look cute, handsome, winsome, clever, beautiful, elegant, grungy, powerful, tough, scary, lithe, bouncy, dashing, and a string of other adjectives that supposedly put them above the stratosphere of ordinary. What are some dramatic and telling features that will draw and hold the player's attention for level after level of gameplay? What would attract your player to the screen?

Look at your character's personality traits for clues. If you wrote down "lazy" as one of your character's personality traits, what are some visual cues you'd use to reveal that he is lazy? Maybe you'd give him eyelids at half-mast, rumpled clothing, and scruffy shoes. Whatever you do, create a crafted visual hook by selecting one or more specific traits about your character and accentuating them. Ash Ketchum has his blue vest, green gloves, and cap. Mario has his black mustache, red cap, and overalls. Indiana Jones has his fedora and bullwhip. Spock has his neatly trimmed hair and pointy ears and eyebrows.

Five Lessons of Designing Avatars

If you want to be a successful game designer, it is important to learn from the hard-earned lessons of others who have come before you. There are five lessons

concerning the design of game avatars you should take to heart, and they are as follows:

1. Capitalize on lucky accidents
2. Exaggerate action
3. Use appealing colors
4. Use your own art style
5. Stay away from stereotypes

Capitalize on Lucky Accidents Shigeru Miyamoto of Nintendo fame said that most of the character quirks he created that made the characters memorable were purely accidental. The only reason Mario got a mustache was that in the old 8-bit graphics it was difficult to tell the difference between Mario's

Figure 6.19
Shigeru Miyamoto, a legend among game designers everywhere, gave Mario a mustache just to separate his nose and chin—and now Mario just wouldn't be Mario without it!

nose and jaw, and so Miyamoto told his artists to add a mustache to separate the two. (See Figure 6.19.) Today, Mario would simply not be Mario without his famed mustache!

Exaggerate Action Shigeru Miyamoto had other industry lessons to share, and the next was exaggeration. Players respond better to game avatars with exaggerated animations. For instance, Mario is short but can easily jump three times his own height. He squishes a little when he falls to the ground. And his spurts of "Woo-hoo!" and "Let's-a-go!" have delighted players for decades. This does not mean every character you make has to be a cartoon character, but adding exaggerated animations will make your game more dramatic and appealing.

Use Appealing Colors Miyamoto said the number one thing he wanted Nintendo artists to remember when designing memorable avatars was color. He wanted strong primary and tertiary color palettes, with high contrast and strong visual staying power, so that the characters were even recognizable as tiny blurs on the screen.

Pick a color palette that appeals to you visually. If you have to, look at a painting or photograph that has an excellent color composition, and then sample it as you paint your character in your image editing software (in Adobe Photoshop or Illustrator you can use the Color Picker window to sample colors like this).

Use Your Own Art Style As every major talent in the world will tell you, impersonation is the sincerest form of flattery, but don't impersonate me! Plagiarism is rampant in the new Internet age that we find ourselves struggling in. Don't shirk from becoming renowned for expressing yourself through your game. Just because you want your game to look like *World of WarCraft* and appear uber-realistic with photographic textures doesn't mean you should. There are lots of game companies out there attempting to accomplish that very same goal. So why not go off on a tangent? Why not do what *you* are good at?

Tim Schafer, creator of *Grim Fandango* and *Psychonauts*, says that the original artistic style and unusual visual flair he imparted on his games led to the success and audience recognition of his games. This line of thought is what has made Tim Schafer, as well as film director Tim Burton, successes in their respective industries.

Do you have a particular art style you could apply to your game characters? Let your creativity out (see Figure 6.20) and get noticed!

Figure 6.20
Use your own wild and crazy art style distinctively its own to get recognition.

Stay Away from Stereotypes Stereotypes are very limiting when it comes to storytelling, and using them damages your respectability when critics attempt to take your work seriously. When creating characters it is never advisable to use stereotypes.

One of the worst stereotypes in video games and comics is the Big-Breasted Bimbo (as seen in Figure 6.21). Big-Breasted Bimbos are apparent in games such as *Tomb Raider* (Lara Croft) and *Bloodrayne* (Rayne). They have exaggerated assets that target juvenile male audiences but do little to engender play for the rest of us. In fact, there are many females that report games like these initially scared them away from gaming altogether and reflect poorly on the industry as a whole.

Other stereotypes to avoid would be growling military generals chomping cigars, shifty-eyed thieves in masks, mad scientists, and studious bookworms. Unless you're poking fun and using them as parody, don't use such obvious stereotyped characters!

Figure 6.21
This is an example of the stereotypical female game character.

Character Animation

There's a common misconception that because digital artists use a computer to do their dirty work, it's quicker or easier. Let's shoot that notion down right now. Computers *do* make certain tasks easier, but creating digital art and animating characters is still deliberately slow and tedious. You have to own a lot of patience to work in the digital art field.

There's this really great saying in the animation field that says, ''Moving stuff around is ***not*** animation!'' Making an animated character is not quick or easy, as I have already explained. If you just want to move objects around on your computer screen, then yes, that *is* quick and easy. You can think yourself a great artist and wait for the money to pour in. But that is not going to happen. Anyone can learn the basics of an animation software application and move stuff around, but no one is going to buy a product anyone can do. They want what *you* can do, or what you will someday be capable of, which is real animation.

Animation is all about bringing characters to life in a virtual world. Life is never easy, so why should replicating it in two-dimensional space be?

Disney Principles of Animation There are principles of animation first set down by Frank Thomas and Ollie Johnston in their book *The Illusion of Life: Disney Animation*, which was edited and authenticated in the 1930s by Disney Studios. These principles apply to all forms of animation, and they are still as useful today as they were then.

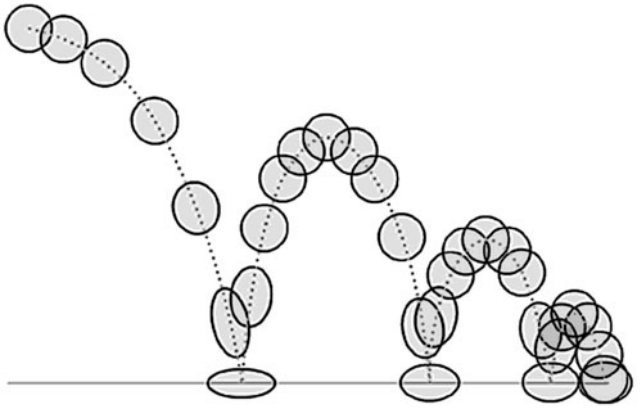


Figure 6.22
The bouncing ball, seen in slow motion, reveals the secrets behind animating object motion and weight distribution.

Most of these principles look at every virtual object as having a body and weight, which physics may be applied against. The most important principle covered is that of motion and weight distribution, exemplified by the "rubber ball" exercise (Figure 6.22). Essentially, whenever an object hits a larger object, it squashes a little, and when it is flying along an arcing motion, it stretches out. Though the volume of the moving object stays the same, its shape can change to fit the motion.

Another of Thomas and Johnston's principles governs anticipation and follow-through actions. The easiest way to understand this is to imagine a basketball player. To make a free-throw shot, like in Figure 6.23, the player bends his knees slightly in anticipation of the action, coiling his arms at the base of the ball like a finely focused spring, and then releasing by thrusting his energy up and out toward the basketball goal. Even after the ball has left the player's hands, the player's arms continue outward to point in an arc toward the goal, as his feet touch the ground again and recoil by the legs bending at the knees.

Overall, the animation principle you should take home with you is motion emphasis. *Motion emphasis* is literally the process of slowing down and over-exaggerating the character action you want performed in order for audiences to keep up with and understand what is happening. For instance, if you've ever been to a karate championship, you know that some of the moves can be so lightning fast, legs and fists so stiff and speedy, that you can barely keep up with what is happening. On the other hand, if you've ever watched a Jackie Chan movie, you can see how he and the other stunt performers slow down the action and exaggerate each motion so the viewers can keep up with what's going on.

Figure 6.23
Several animation principles come into play with a simple free-throw shot.

Animating Motion Cycles Most animations game designers make will be looping ones, which means that once the animation gets from frame one to frame thirty (or whatever the final frame is), it will go back to frame one and repeat as long as necessary. This type of animation is called a *motion cycle*, and the most common motion cycles seen in video games are walk cycles, attack cycles, jump cycles, and so on.

In the last chapter you previewed the flour sack's walk cycle. The character basically walks on two legs similar to how a human would, moving one foot in front of the other. The artist used the way human beings walk, like in Figure 6.24,

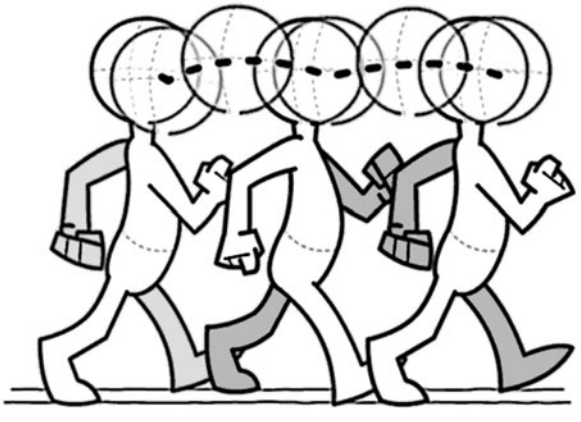


Figure 6.24
The basic walk cycle is an exaggeration of the way a human walks.

as a reference for developing the flour sack's walk cycle. You could do the same with any character, including aliens, robots, and dogs with rocket packs strapped to their backs.

To animate a character for Torque, you must make sure that each cell of animation is plain white and the same size as every other cell. To set up "cells" in Adobe Photoshop you can use the ruler and make guides to outline cell dimensions before placing your objects or characters. You can also trace over, or use original animations provided with Torque Game Builder, to make your own. Though this might be a time-consuming process, having animated entities in your game levels will strengthen visual interest.

Further Animation Resources Animation is a long, complicated topic and too involved to get into in any further detail here. However, you can learn more about animation from these guides:

- *The Animator's Survival Kit* by Richard Williams, published by Faber and Faber, Ltd.
- *Cartoon Animation* by Preston Blair, published by Walter Foster Publishing, Ltd.
- *The Animator's Reference Book* by Les Pardew and Ross Wolfley, published by Course Technology.

Making an Animated Character in TGB

For the purposes of our exercise, an animated player avatar has already been created. Named Batty (he's a bat), you will find he has been animated so that he will look like he is flapping his wings. The following instructions will help you bring your player character, Batty, into the game level.

1. Go to the Create tab on the right-hand side of the editor.
2. Click the Create a New Image Map button under the tab.
3. Locate the Batty.png file. Click OK and Batty will appear under your Static Sprites.
4. Double-click on the Batty thumbnail under Static Sprites to open the Image Builder dialog window. Change the Image Mode to CELL. Cell mode is used for images that need to be evenly divided into more than

one frame. The default Cell setting is four equally sized frames. This is perfect for Batty, so we won't have to change a thing. If it was wrong, all we'd have to do is change the Cell Width and Cell Height to match our image.

5. Click Save.
6. Under the Create tab, click on Create a new Animation.
7. Select BattyImageMap as the source image.
8. Click the Add All Frames from Image green plus sign button and then click the Play Animation preview button to see how Batty will look animated. See him flapping like mad?
9. Change the Frames Per Second to a value of 8 and press Enter. Now Batty won't look like a hummingbird! Compare your work to Figure 6.25.
10. Click Save to exit the Animation Builder window.
11. To place the sprite into your game, simply drag-and-drop the Batty image from the Animated Sprites section into the center of your level.

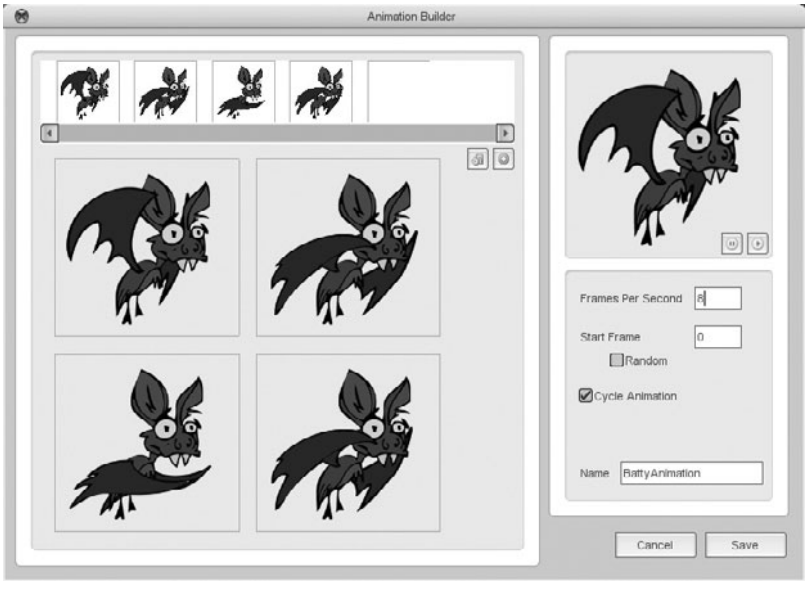


Figure 6.25
You should see a preview of Batty appear on the right.

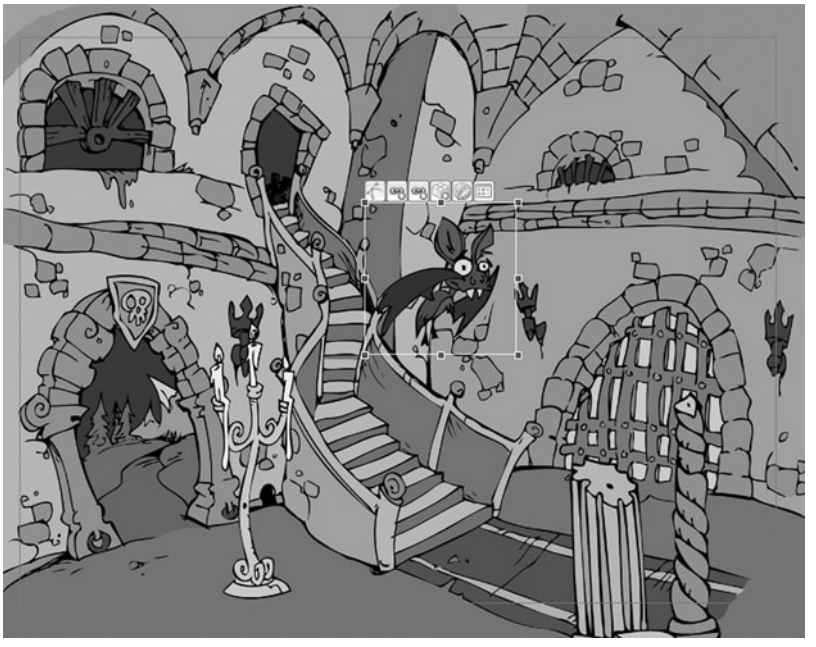


Figure 6.26
Set Batty into your level and resize him so that he looks okay.

12. Now hold down the Shift key as you click and drag one of the corner handles of the bat, and resize him to appear an appropriate size for our level (like in Figure 6.26).

13. It's time to set Batty's layer. We have the background image set to 30, the candelabra to 20, the pillars at 6, and the candle flames at 19. We want to place Batty somewhere in-between 19 and 6, so that he can walk in front of the candles but behind the pillars. Click the Edit tab and scroll down to Batty's Layer setting. Input 10 and hit Enter, or press the right arrow 10 times to get the desired value.

Programming Batty to Fly

To make Batty fly, we need to add some programming code.

To do this, browse out to your MyCastle\Game\GameScripts folder. Inside this folder you should see a game.cs file. Any time you see a file with .cs on the end of it as the extension, you are looking at a TorqueScript file. If you're using Microsoft, open up the game.cs file by right-clicking on it and selecting Open With > Notepad (or Wordpad). If you're using Macintosh, you can open it by

holding down the CMD key and clicking on the game.cs file; then choose Open With > TextEdit. This will open up your TorqueScript for your game inside your operating system's text editor.

Optionally, you could use Torsion or another syntax editor described in Chapter 5, ''Programming with TorqueScript.''

Note

You will be working with several files with the same name but different file extensions. For instance, the main.cs is a code file you can edit, while the main.cs.dso file is a backup file that streamlines loading speed. If your PC is not set up to view file extensions, you must set it up before beginning game construction. In your Explorer window, go to Tools > Folder Options to bring up the Folder Options dialog window. In this window, click on the View tab and scroll down until you find the Hide Extensions for Known File Types checkbox. Uncheck that option and click the OK button to exit the Folder Options dialog window. You should now be able to see all file extensions, making it easier to complete the exercises in this book.

You should see the following data already in your game.cs file:

```
//--------------------------------------------------------------------------
-----------------
// Torque Game Builder
// Copyright (C) GarageGames.com, Inc.
//--------------------------------------------------------------------------
-----------------

//--------------------------------------------------------------------------
-----------------
// startGame
// All game logic should be set up here. This will be called by the level builder
when you
// select "Run Game" or by the startup process of your game to load the first level.
//--------------------------------------------------------------------------
-----------------
function startGame(%level)
{
  Canvas.setContent(mainScreenGui);
  Canvas.setCursor(DefaultCursor);

  new ActionMap(moveMap);
  moveMap.push();
```

```
  $enableDirectInput = true;
  activateDirectInput();
  enableJoystick();

  sceneWindow2D.loadLevel(%level);
}

//--------------------------------------------------------------------------
-----------------
// endGame
// Game cleanup should be done here.
//--------------------------------------------------------------------------
-----------------
function endGame()
{
  sceneWindow2D.endLevel();
  moveMap.pop();
  moveMap.delete();
}
```

This script file is pretty basic. It is the default function called whenever we test our level from the Level Builder. When you click the Play Scene button, the `startGame()` function is called, and any subroutines within it are also dynamically called.

Create a New Class We will use script classes to assimilate Batty from the Level Builder into our program. To do this we will create a class for Batty and then, in the Level Builder, we will assign the Batty class to our Batty image. Once Batty is its own class, we can automatically get a whole subroutine called to handle just Batty whenever the level is loaded.

Create a new text file in MyCaste\Games\GameScripts folder where you found game.cs. Name this new text file player.cs—making sure to add the .cs so that the game engine will read it as a script file. Open the player.cs file in your default text editor and type this snippet:

```
function BattyPlayer::onLevelLoaded(%this, %scenegraph)
{

}
```

Let's break this down. We start this function with the keyword `function`, which tells the engine that we're beginning a function declaration. Then we follow this by a class name; in this case, we invent one called `BattyPlayer`. Since we'll be attaching the `BattyPlayer` to the Batty image in our level later, this function will be tagged to any entity given the `BattyPlayer`, in this case meaning the Batty image. The code `onLevelLoaded` is the function name that will be called on the object, which in this circumstance means this function is called when the level first loads. The `%this` separates the object with this class into a separate instance, just in case there are multiple objects sharing the same class name. The `%scenegraph` identifies the object that holds everything in our level.

Make Batty Fly Now we are going to set our `onLevelLoaded` so that we get Batty flying. We will start by preparing Batty as a global variable. Remember that a global variable is shown with a "$" in front of it, and it can be used elsewhere outside the main script, but a local variable, like `%this`, is shown with a "%" in front of it, and it is only temporary.

Add the following line between the curly braces of our `Batty` class script:

```
function BattyPlayer::onLevelLoaded(%this, %scenegraph)
{
 $MeBatty = %this;
}
```

Now we want to add movement commands, so that the player can control Batty.

```
function BattyPlayer::onLevelLoaded(%this, %scenegraph)
{
 $MeBatty = %this;
 moveMap.bindCmd(keyboard, "w", "BattyPlayerUp();", "BattyPlayerUpStop();");
 moveMap.bindCmd(keyboard, "s", "BattyPlayerDown();",
"BattyPlayerDownStop();");
 moveMap.bindCmd(keyboard, "a", "BattyPlayerLeft();",
"BattyPlayerLeftStop();");
 moveMap.bindCmd(keyboard, "d", "BattyPlayerRight();",
"BattyPlayerRightStop();");
}
```

The `moveMap` object is a handle we use for key events when running our level. The `bindCmd` is used in conjunction with the `moveMap` handler to pass it values. First, we specify the exact key we are mapping, and then we declare what function we want to call when that key is pressed. The final declaration tells the computer what subroutine to call when the key is released, or not being pressed any longer. For

instance, we tell the computer that when the W key is pressed that `BattyPlayerUp()` is called, which makes the player avatar move up the screen, and when the W key is no longer pressed that `BattyPlayerUpStop()` is called, which makes the player avatar stop moving up the screen.

Unfortunately, these subroutines don't exist yet. So we are going to have to program them.

After the close of the last curly brackets ({}) of your `onLevelLoaded()` function, insert this new function:

```
function BattyPlayerUp()
{
  $BattyPlayer.setLinearVelocityY(-15);
}
```

This sets Batty to move up the screen whenever the function is called, which we've already mapped to the W key. First of all we choose one of two coordinates used on computer screens: X or Y. X runs horizontally from left to right across the screen, so think of X as the horizontal line. Y runs vertically up and down the screen, so think of Y as the vertical line. In this function we set the `Batty` object to move up 15 increments along the Y axis. Numbers along the Y axis get lower the higher you go up, and get bigger the lower down the screen you go, which is why we made 15 increments a negative number. If we'd used a positive number for Y, it would have made the object move down the screen, which is the opposite of what we want.

Now you need to add three more functions for Batty's movement.

```
function BattyPlayerDown()
{
  $BattyPlayer.setLinearVelocityY(15);
}
function BattyPlayerLeft()
{
  $BattyPlayer.setLinearVelocityX(-25);
}
function BattyPlayerRight()
{
  $BattyPlayer.setLinearVelocityX(25);
}
```

These functions work the same way as the first one. Note that we switch, when moving the character horizontally, from using the Y axis to using the X axis. Also note that to move the player character left across the screen we have to use a negative number. That is because the farther left you go on the X axis, the lower the number will be, and the farther right you go on the X axis, the higher the number will be.

Lastly you have to program Batty's stop movement functions. So add these to the end of your script:

```
function BattyPlayerUpStop()
{
  $BattyPlayer.setLinearVelocityY(0);
}
function BattyPlayerDownStop()
{
  $BattyPlayer.setLinearVelocityY(0);
}
function BattyPlayerLeftStop()
{
  $BattyPlayer.setLinearVelocityX(0);
}
function BattyPlayerRightStop()
{
  $BattyPlayer.setLinearVelocityX(0);
}
```

We are setting the up and down functions' Y velocity to zero and the left and right functions' X velocity to zero. This causes the `Batty` object to stop moving the moment the key mapped to the above functions is released. So when you want to move Batty right across the screen, you press the D key, but as soon as you let up on the D key, the `Batty` object will stop in place.

Your player.cs file should read like this now:

```
function BattyPlayer::onLevelLoaded(%this, %scenegraph)
{
 $MeBatty = %this;
 moveMap.bindCmd(keyboard, "w", "BattyPlayerUp();", "BattyPlayerUpStop();");
 moveMap.bindCmd(keyboard, "s", "BattyPlayerDown();",
"BattyPlayerDownStop();");
 moveMap.bindCmd(keyboard, "a", "BattyPlayerLeft();",
"BattyPlayerLeftStop();");
```

```
 moveMap.bindCmd(keyboard, "d", "BattyPlayerRight();",
"BattyPlayerRightStop();");
}

function BattyPlayerUp()
{
  $BattyPlayer.setLinearVelocityY(-15);
}

function BattyPlayerDown()
{
  $BattyPlayer.setLinearVelocityY(15);
}

function BattyPlayerLeft()
{
  $BattyPlayer.setLinearVelocityX(-25);
}

function BattyPlayerRight()
{
  $BattyPlayer.setLinearVelocityX(25);
}

function BattyPlayerUpStop()
{
  $BattyPlayer.setLinearVelocityY(0);
}

function BattyPlayerDownStop()
{
  $BattyPlayer.setLinearVelocityY(0);
}

function BattyPlayerLeftStop()
{
  $BattyPlayer.setLinearVelocityX(0);
}

function BattyPlayerRightStop()
{
  $BattyPlayer.setLinearVelocityX(0);
}
```

Done! Save your player.cs.

Now you have to link the player.cs file to the game.cs file, so that when the engine reads the game.cs file it will go out and find (and also read) the player.cs file. Open the game.cs file in your default text editor, if you don't still have it open already, and modify it as follows:

```
//--------------------------------------------------------------------------
-----------------
// Torque Game Builder
// Copyright (C) GarageGames.com, Inc.
//--------------------------------------------------------------------------
-----------------

//--------------------------------------------------------------------------
-----------------
// startGame
// All game logic should be set up here. This will be called by the level builder
when you
// select "Run Game" or by the startup process of your game to load the first level.
//--------------------------------------------------------------------------
-----------------
function startGame(%level)
{
  exec("./player.cs");

  Canvas.setContent(mainScreenGui);
  Canvas.setCursor(DefaultCursor);

  new ActionMap(moveMap);
  moveMap.push();

  $enableDirectInput = true;
  activateDirectInput();
  enableJoystick();

  sceneWindow2D.loadLevel(%level);
}
```

```
//--------------------------------------------------------------------------
-----------------
// endGame
// Game cleanup should be done here.
//--------------------------------------------------------------------------
-----------------
function endGame()
{
  sceneWindow2D.endLevel();
  moveMap.pop();
  moveMap.delete();
}
```

All you're really doing is placing the command `exec("./player.cs");` at the beginning of the `startGame` function. This calls in the entire script we'd just written for use in our game.cs file, which is exactly what we want. Now you're almost ready to test your program!

Flight Test Before you can test, you will have to set the Batty image in your level to the `Batty` class. That way, when the level is played, the script we just created is called dynamically.

Back in your Level Builder, select your Batty image in the scene and click the Edit tab on the right. Scroll down and expand the Scripting rollout section to find the Class field. Enter **BattyPlayer** in the Class field and press Enter.

Save your level by clicking the Save icon button (shown in Figure 6.27). Click the Play Scene icon button (shown in Figure 6.27) to test your level. Now, try pressing our movement keys (A, S, W, and D) and you should see the bat image being moved in the appropriate direction. If you find an error, or Batty is reacting inappropriately, you might have to go back to your player.cs file and

Figure 6.27
The Save icon button (left) and the Play Level icon button (right).

double-check it for proper syntax and spelling. The most common mistakes in programming usually boil down to misspellings or leaving out a semicolon. Always proofread your scripts before getting mad at the computer!

Setting Speed Variables To start with, we have to take out the hardcoded velocity ranges. No longer can we determine his velocity by a set value. So open up player.cs in your text editor and edit your movement functions to look like this:

```
function BattyPlayerUp()
{
  $BattyPlayer.setLinearVelocityY(-$BattyPlayer.vSpeed);
}
function BattyPlayerDown()
{
  $BattyPlayer.setLinearVelocityY($BattyPlayer.vSpeed);
}
function BattyPlayerLeft()
{
  $BattyPlayer.setFlipX(true);
  $BattyPlayer.setLinearVelocityX(-$BattyPlayer.hSpeed);
}
function BattyPlayerRight()
{
  $BattyPlayer.setFlipX(false);
  $BattyPlayer.setLinearVelocityX($BattyPlayer.hSpeed);
}
```

After finishing, save your player.cs file.

Notice that the left and right movement functions have had the speed values replaced with a reference to `hSpeed`, which will be a horizontal speed we will store on the `Batty` object. The up and down movement functions have had the speed values replaced with a reference to `vSpeed`, which will be a vertical speed we will store on the `Batty` object.

Now we have to create the `hSpeed` and `vSpeed` variables and set their default values within the Level Builder, so return to the Level Builder.

Select your bat image in the scene and go to the Edit tab. Scroll down and expand the Dynamic Fields rollout.

The Field Name input field is the name of the new value we want to set, and the Field Value input field is the actual number we want to store for the value. We

want to store two new values for our object, `hSpeed` and `vSpeed`. The following are the steps you have to take:

1. First, enter **`hSpeed`** for Field Name and **25** for Field Value. Then click the little green plus sign button to confirm that you want to add that value.
2. Next enter **`vSpeed`** for Field Name and **15** for Field Value. Click the little green plus sign button to confirm that you want to add that value.
3. Save and play your level. Batty should move at the same speeds he did before. If he doesn't, double-check your program script and that you entered the values in his object properties correctly.

Make Batty Fly Right You might have noticed that when you move Batty right and left he always faces the same direction. Fixing this is very simple. You just have to add one line of code to both the left and right movement functions you created above.

So open your player.cs file in your text editor. Scroll down to the function `BattyPlayerRight()` and edit it so that it looks like this:

```
function BattyPlayerRight()
{
  $BattyPlayer.setFlipX(false);
  $BattyPlayer.setLinearVelocityX($BattyPlayer.hSpeed);
}
```

We set `FlipX` to "false" so that Batty will face the default direction when moving right. Now we have to set `FlipX` to "true" so that when Batty flies the opposite way, to the left, he'll turn around and face that direction. Change the function `BattyPlayerLeft()` to look like this:

```
function BattyPlayerLeft()
{
  $BattyPlayer.setFlipX(true);
  $BattyPlayer.setLinearVelocityX(-$BattyPlayer.hSpeed);
}
```

Be sure to save your player.cs file and go back to the Level Builder and test it. Press the D key to move right and observe what happens; then press the A key to move left and see how Batty flips around. Done!

Adding Automatic Updates Right now Batty's movement is fine but somewhat jerky. Every time a key is released, even if another is being pressed, he

immediately halts at a dead stop. This is because our code says that when a key bind is released, his overall velocity resets to zero. This stops him dead in his tracks, which gets annoying fast.

To prevent this, we will have to set up a conditional logic algorithm that checks to see what keys are being pressed and only makes Batty freeze in place if there is no input.

Open up your player.cs script file in your default text editor. Underneath the `onLevelLoaded()` function, add this new function:

```
function BattyPlayer::updateMovement(%this)
{
  if(%this.moveUp)
  {
   $BattyPlayer.setLinearVelocityY(-$BattyPlayer.vSpeed);
  }
   if(%this.moveDown)
  {
   $BattyPlayer.setLinearVelocityY($BattyPlayer.vSpeed);
  }
 if(%this.moveLeft)
  {
   $BattyPlayer.setFlipX(true);
   $BattyPlayer.setLinearVelocityX(-$BattyPlayer.hSpeed);
  }
  if(%this.moveRight)
  {
   $BattyPlayer.setFlipX(true);
   $BattyPlayer.setLinearVelocityX($BattyPlayer.hSpeed);
  }
  if(!%this.moveLeft && !%this.moveRight)
  {
   %this.setLinearVelocityX(0);
  }
  if(!%this.moveUp && !%this.moveDown)
  {
   %this.setLinearVelocityY(0);
  }
}
```

The selective logic of this statement should seem pretty straightforward. For the most part, you are telling the computer to check to see if the player is pressing

keys, and if so move the character. The last two subroutines might seem a little strange to you at first, but basically they check to see that neither the left nor right key is being pressed, and if not, return a zero velocity for the X axis. It also checks to see if the up or down keys are being pressed, and if they aren't, returns a zero velocity for the Y axis.

Now we have to replace the movement functions we coded earlier. Edit them to look like this:

```
function BattyPlayerUp()
{
  $BattyPlayer.moveUp = true;
  $BattyPlayer.updateMovement();
}
function BattyPlayerDown()
{
  $BattyPlayer.moveDown = true;
  $BattyPlayer.updateMovement();
}
function BattyPlayerLeft()
{
  $BattyPlayer.moveLeft = true;
  $BattyPlayer.updateMovement();
}
function BattyPlayerRight()
{
  $BattyPlayer.moveRight = true;
  $BattyPlayer.updateMovement();
}
```

And you have to replace the movement stops as well:

```
function BattyPlayerUpStop()
{
  $BattyPlayer.moveUp = false;
  $BattyPlayer.updateMovement();
}
function BattyPlayerDownStop()
{
  $BattyPlayer.moveDown = false;
  $BattyPlayer.updateMovement();
}
function BattyPlayerLeftStop()
```

```
{
  $BattyPlayer.moveLeft = false;
  $BattyPlayer.updateMovement();
}
function BattyPlayerRightStop()
{
  $BattyPlayer.moveRight = false;
  $BattyPlayer.updateMovement();
}
```

If you can't tell, the new movement functions set the action to `true` so that the `updateMovement()` function will read the input correctly, and the new stop functions set the same action to false so that `updateMovement()` will reduce velocity to zero. This tweak will cause a smoother character motion in the game. Try it out now! Be sure to save your player.cs file, then return to the Level Builder and click the Play Level icon button. You should see the jerky start-and-stop motions have been removed, and Batty is flying like a pro! If it doesn't play right, go back to your player.cs file and double-check it for errors.

So far, player.cs should read like this:

```
function BattyPlayer::onLevelLoaded(%this, %scenegraph)
{
 $MeBatty = %this;
 moveMap.bindCmd(keyboard, "w", "BattyPlayerUp();", "BattyPlayerUpStop();");
 moveMap.bindCmd(keyboard, "s", "BattyPlayerDown();",
 "BattyPlayerDownStop();");
 moveMap.bindCmd(keyboard, "a", "BattyPlayerLeft();",
 "BattyPlayerLeftStop();");
 moveMap.bindCmd(keyboard, "d", "BattyPlayerRight();",
 "BattyPlayerRightStop();");
}

function BattyPlayer::updateMovement(%this)
{
  if(%this.moveUp)
  {
   $BattyPlayer.setLinearVelocityY(-$BattyPlayer.vSpeed);
  }
   if(%this.moveDown)
  {
   $BattyPlayer.setLinearVelocityY($BattyPlayer.vSpeed);
  }
```

```
 if(%this.moveLeft)
  {
   $BattyPlayer.setFlipX(true);
   $BattyPlayer.setLinearVelocityX(-$BattyPlayer.hSpeed);
  }
  if(%this.moveRight)
  {
   $BattyPlayer.setFlipX(true);
   $BattyPlayer.setLinearVelocityX($BattyPlayer.hSpeed);
  }
  if(!%this.moveLeft && !%this.moveRight)
  {
   %this.setLinearVelocityX(0);
  }
  if(!%this.moveUp && !%this.moveDown)
  {
   %this.setLinearVelocityY(0);
  }
}

function BattyPlayerUp()
{
  $BattyPlayer.setLinearVelocityY(-$BattyPlayer.vSpeed);
}

function BattyPlayerDown()
{
  $BattyPlayer.setLinearVelocityY($BattyPlayer.vSpeed);
}

function BattyPlayerLeft()
{
  $BattyPlayer.setFlipX(true);
  $BattyPlayer.setLinearVelocityX(-$BattyPlayer.hSpeed);
}

function BattyPlayerRight()
{
  $BattyPlayer.setFlipX(false);
  $BattyPlayer.setLinearVelocityX($BattyPlayer.hSpeed);
}

function BattyPlayerUp()
```

```
{
  $BattyPlayer.moveUp = true;
  $BattyPlayer.updateMovement();
}
function BattyPlayerDown()
{
  $BattyPlayer.moveDown = true;
  $BattyPlayer.updateMovement();
}
function BattyPlayerLeft()
{
  $BattyPlayer.moveLeft = true;
  $BattyPlayer.updateMovement();
}
function BattyPlayerRight()
{
  $BattyPlayer.moveRight = true;
  $BattyPlayer.updateMovement();
}

function BattyPlayerUpStop()
{
  $BattyPlayer.moveUp = false;
  $BattyPlayer.updateMovement();
}

function BattyPlayerDownStop()
{
  $BattyPlayer.moveDown = false;
  $BattyPlayer.updateMovement();
}

function BattyPlayerLeftStop()
{
  $BattyPlayer.moveLeft = false;
  $BattyPlayer.updateMovement();
}

function BattyPlayerRightStop()
{
  $BattyPlayer.moveRight = false;
  $BattyPlayer.updateMovement();
}
```

Notice how most of this programming stuff is iterative, meaning that you do it in steps and construct your code so that you can go back and add to it later? You start with a basic foundation script and then keep tweaking and adding to it until it runs smoother.

Adding a Speed Boost Batty is moving just fine, but we want to give him an occasional speed boost. Speed boosts can come from items the player gains during play, or be limited by a timer. For our intents right now, we'll map a speed boost to the spacebar. Whenever the player presses the spacebar, Batty will take off like a little furry mammal rocket.

Open up your player.cs script file. Right now your `onLevelLoaded()` function should look like this:

```
function BattyPlayer::onLevelLoaded(%this, %scenegraph)
{
 $MeBatty = %this;
 moveMap.bindCmd(keyboard, "w", "BattyPlayerUp();", "BattyPlayerUpStop();");
 moveMap.bindCmd(keyboard, "s", "BattyPlayerDown();",
 "BattyPlayerDownStop();");
 moveMap.bindCmd(keyboard, "a", "BattyPlayerLeft();",
 "BattyPlayerLeftStop();");
 moveMap.bindCmd(keyboard, "d", "BattyPlayerRight();",
 "BattyPlayerRightStop();");
}
```

You want to add one more key command at the end, just after the last `moveMap.bindCmd()` call and before the closing curly bracket:

```
moveMap.bindCmd(keyboard, "space", "BattyPlayerBoost();",
"BattyPlayerBoostStop();");
```

This is designed the same way our previous key mapping binds were done. We are binding the spacebar to execute `BattyPlayerBoost()` when pressed and `BattyPlayerBoostStop()` when released. Now we have to write both of these functions into our code, so scroll down to the very bottom of player.cs and add the following:

```
function BattyPlayerBoost()
{
  %flipX = $BattyPlayer.getFlipX();
  if(%flipX)
  { %hSpeed = -$BattyPlayer.hSpeed * 3; }
```

```
  else
  { %hSpeed = $BattyPlayer.hSpeed * 3; }
  $BattyPlayer.setLinearVelocityX(%hSpeed);
}
```

This function is pretty busy, and clever as well. For one thing, it determines the direction the player character is currently moving by pulling the `flipX` out of our bat object. If Batty is facing right, we know that `flipX` will be false, and if he's facing left, we know that `flipX` will be true. If Batty's moving left, we want the boost to store a negative value that is three times his current `hSpeed`—and if he's moving right, we want the boost to store a positive value that is three times his current `hSpeed`. This way when the player presses the spacebar, Batty should zip around at triple his normal speed.

Now put this after that function:

```
function BattyPlayerBoostStop()
{
  $BattyPlayer.setLinearVelocityX(0);
}
```

This will turn the boost off as soon as the spacebar is released. Actually, it stops Batty in his tracks the second the spacebar is depressed, because it returns a zero forward velocity. In a way, this balances the effectiveness of the speed boost. The player can have Batty leap forward, but doing so will cause him to stop in his tracks when it's through.

Save your player.cs file and go back to the Level Builder. Play your level, and try out your speed boost by occasionally pressing and releasing the spacebar. If it does not work correctly, go back to your program script and proofread it carefully.

Set Batty's World Limit Batty can move in any direction now, but we don't want him to run off the edge of the screen, because the player cannot see his character any longer if he does that. We change this by setting a world limit. A *world limit* is a bounding box we define visually in the Level Builder and sets the world bounds so a character can't disappear off-screen. Let's define Batty's world limit.

1. In the Level Builder, select the Batty image and roll over it with your mouse cursor. You will see a set of icons appear above the image, as seen in Figure 6.28. Click the second icon from the right, which is the World Limit tool.

Figure 6.28
Find the World Limit Tool icon in the set of icons appearing above your image.

2. Your view will change until it is zoomed way out from your scene. Plus, you should see a scalable box representing the world limit of your entity, like in Figure 6.29. At first the limit far exceeds the scene. Scale the box so that the left and right sides match the edges of your camera view.

3. Once you've set the world limit boundary, click the Selection tool in the top toolbar (see Figure 6.30) to exit the world limit editing mode.

4. Click the Edit tab and scroll down to find the World Limits rollout. The default is set to OFF but you want to change it to CLAMP. There are plenty of prefabricated responses, but CLAMP should allow us to stop Batty from running out of the camera view.

5. Save your level before clicking the Play Level icon button. Try to move your Batty character to the rightmost border, where previously he'd run off the screen. Now he'll be stopped appropriately. Done!

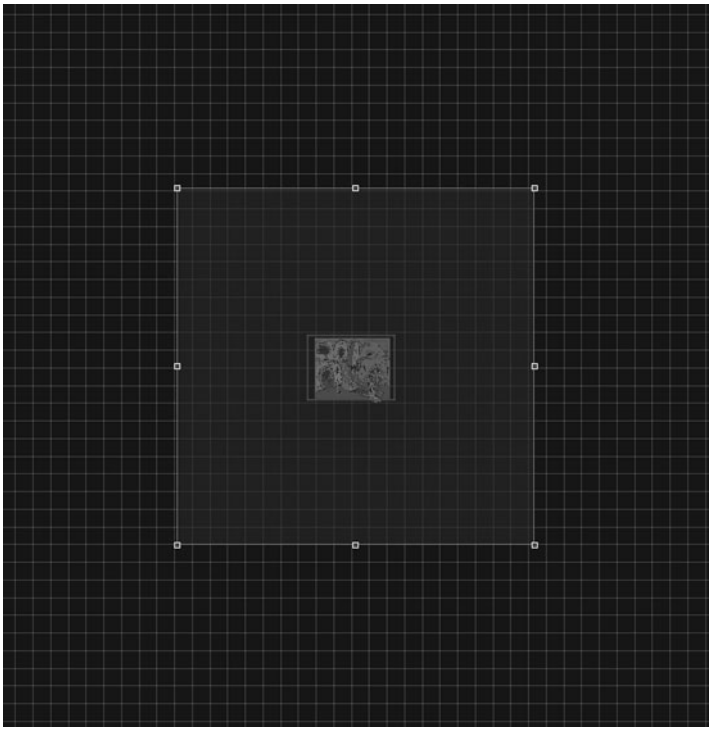


Figure 6.29
The resizable box that represents the world limit.

Figure 6.30
The Selection tool is found in the top toolbar.

Proofreading Your Code You should have modified your game.cs file to execute your player.cs file. Your player.cs file, when done with this exercise, should read as such:

```
function BattyPlayer::onLevelLoaded(%this, %scenegraph)
{
 $MeBatty = %this;
 moveMap.bindCmd(keyboard, "w", "BattyPlayerUp();", "BattyPlayerUpStop();");
 moveMap.bindCmd(keyboard, "s", "BattyPlayerDown();",
 "BattyPlayerDownStop();");
```

```
 moveMap.bindCmd(keyboard, "a", "BattyPlayerLeft();",
 "BattyPlayerLeftStop();");
 moveMap.bindCmd(keyboard, "d", "BattyPlayerRight();",
 "BattyPlayerRightStop();");
 moveMap.bindCmd(keyboard, "space", "BattyPlayerBoost();",
 "BattyPlayerBoostStop();");
}

function BattyPlayer::updateMovement(%this)
{
  if(%this.moveUp)
  {
   $BattyPlayer.setLinearVelocityY(-$BattyPlayer.vSpeed);
  }
   if(%this.moveDown)
  {
   $BattyPlayer.setLinearVelocityY($BattyPlayer.vSpeed);
  }
 if(%this.moveLeft)
  {
   $BattyPlayer.setFlipX(true);
   $BattyPlayer.setLinearVelocityX(-$BattyPlayer.hSpeed);
  }
  if(%this.moveRight)
  {
   $BattyPlayer.setFlipX(true);
   $BattyPlayer.setLinearVelocityX($BattyPlayer.hSpeed);
  }
  if(!%this.moveLeft && !%this.moveRight)
  {
   %this.setLinearVelocityX(0);
  }
  if(!%this.moveUp && !%this.moveDown)
  {
   %this.setLinearVelocityY(0);
  }
}

function BattyPlayerUp()
{
  $BattyPlayer.setLinearVelocityY(-$BattyPlayer.vSpeed);
}
```

```
function BattyPlayerDown()
{
  $BattyPlayer.setLinearVelocityY($BattyPlayer.vSpeed);
}

function BattyPlayerLeft()
{
  $BattyPlayer.setFlipX(true);
  $BattyPlayer.setLinearVelocityX(-$BattyPlayer.hSpeed);
}

function BattyPlayerRight()
{
  $BattyPlayer.setFlipX(false);
  $BattyPlayer.setLinearVelocityX($BattyPlayer.hSpeed);
}

function BattyPlayerUp()
{
  $BattyPlayer.moveUp = true;
  $BattyPlayer.updateMovement();
}
function BattyPlayerDown()
{
  $BattyPlayer.moveDown = true;
  $BattyPlayer.updateMovement();
}
function BattyPlayerLeft()
{
  $BattyPlayer.moveLeft = true;
  $BattyPlayer.updateMovement();
}
function BattyPlayerRight()
{
  $BattyPlayer.moveRight = true;
  $BattyPlayer.updateMovement();
}

function BattyPlayerUpStop()
{
  $BattyPlayer.moveUp = false;
  $BattyPlayer.updateMovement();
}
```

```
function BattyPlayerDownStop()
{
  $BattyPlayer.moveDown = false;
  $BattyPlayer.updateMovement();
}

function BattyPlayerLeftStop()
{
  $BattyPlayer.moveLeft = false;
  $BattyPlayer.updateMovement();
}

function BattyPlayerRightStop()
{
  $BattyPlayer.moveRight = false;
  $BattyPlayer.updateMovement();
}

function BattyPlayerBoost()
{
  %flipX = $BattyPlayer.getFlipX();
  if(%flipX)
  { %hSpeed = -$BattyPlayer.hSpeed * 3; }
  else
  { %hSpeed = $BattyPlayer.hSpeed * 3; }
  $BattyPlayer.setLinearVelocityX(%hSpeed);
}

function BattyPlayerBoostStop()
{
  $BattyPlayer.setLinearVelocityX(0);
}
```

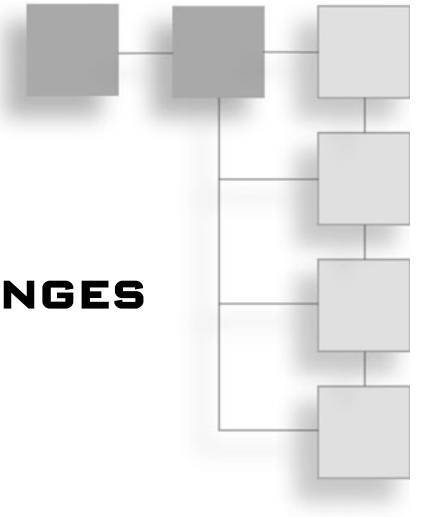
Double check your work, and test frequently for errors. That's it!

Review

After reading this chapter, you should understand the following:

- How to set up your workspace, including camera view and design resolution
- How to import resources for your game project
- Some proven tips on level and character design
- How to make a player character and get it to move around a basic game level

CHAPTER 7



Adding Game Challenges

In this chapter, you will learn:

- The different types of challenges popular among games
- How to add conflict and remain fair when making game challenges
- How to add a reward in your game project and use the TGB collision system
- How to add an obstacle in your game project

So far you've learned the basics of designing 2D games, how to make the graphics for a 2D game, and how to put the art into a level using Torque Game Builder. You've also scripted a player character, which you can then use to build a game around. A game just isn't a game without some clearly defined challenges. The player must be motivated to play the game and to succeed. How she does so is intrinsically linked to how you have set up the challenges. So let's look at game challenges for a minute, and then we'll finish building our first game project.

Game Challenges

Tip

"Our species can only survive if we have obstacles to overcome. You take away all obstacles. Without them to strengthen us, we will weaken and die."

—Captain Kirk, *Star Trek* ("Metamorphosis")

Figure 7.1
This scene reveals just how desperate some people are to find closeout sales.

Most of the time, challenges take the form of obstacles that must be overcome. Either the player faces a horde of hungry zombies shuffling along a grainy windswept city street (or occasional shopping mall, as seen in Figure 7.1) or the player is trapped in a ski lodge during a blizzard while a serial killer erases each of the player's in-game allies one by one. Or something else assaults the player, which requires her definite attention, focus, and decision-making skills to overcome and maybe, just maybe, triumph.

There are two elements responsible in every challenge:

- The hint of reward
- The obstacle to overcome to reach that reward

Players are by no means stupid. They understand what's what, and they know to look for Pavlovian mechanics within a game, primarily (a) what they need to achieve (i.e., a goal, objective, mission directive, or other reward), and (b) what to do to get it. Everything else is just conflict.

Note

When we say Pavlovian, we are talking about Ivan Petrovich Pavlov, the Russian physiologist, psychologist, and physician who won the Nobel Prize in Physiology/Medicine in 1904. He is best known as the guy who first described classical conditioning, a form of associated learning where he showed that animals can begin to associate rewards with the work they have to do to receive them. In other words, when you feed your dog every time you ring a special bell, the dog will begin to associate the bell with getting fed. This is true of humans, and especially video gamers!

Common Types of Game Challenges

The following are the most common types of game challenges you can find:

- **Alchemical**: requires the player to take one or more objects and put them together to form something that they need to use (caution: never assume the player will guess what parts go together to make what!).
- **Clue-driven**: requires the player to find and make use of a crucial bit of info, such as a password, key code, secret, etc., to pass by a guard, locked door, or to open a locked briefcase.
- **Lock mechanism**: a popular device used in-game to prevent player progress until certain obstacles have been overcome or goals have been met. Some lock mechanisms can be as prosaic as the common lock: a locked door, a jammed gateway, or elevator without power. Older arcade games used *blood locks*, which were arbitrary lock mechanisms where the player was rushed by whopping bad guys, and only after every one of the enemies was defeated would the player's path become clear.
- **Mazes**: adding lots of twists, turns, and dead ends quickly makes a standard game level into a maze that is a wonderfully entertaining way to break the monotony of locked doors.
- **Mental deduction**: requires the player to figure out causal relationships between their character and the environment; for instance, having the pathway blocked by a wall that only a certain hammer can bash open is an example of a mental deduction puzzle.
- **Monsters**: block the pathway or they carry the key to our player's survival. Monsters scare and titillate players and form a challenge just by being difficult to tackle.

Figure 7.2
Saw blades and similar traps are often timing/sequence challenges.

- **Rules-bound**: there's an old saying in game design that "players don't beat the game, they beat the system"—meaning that players learn the underlying workings, or rules, of the system they are playing and then become better at finding shortcuts, loopholes, and sure-things. This is true of classic card and board games, as well as video games.
- **Social/dialogue**: the player must encounter non-player characters, or NPCs, and talk to each of them to find some that will help him past his hurdle. The player's choices will often take the form of complex dialogue decisions.
- **Timing/sequence**: requires the player to identify appropriate moves in a sequence of timed events to work around. One example includes the obvious puzzles littered throughout the *Prince of Persia* games, where if you fail to run past the spinning saw blades in time or swing to the next ledge, as seen in Figure 7.2, you die!
- **Traps**: a hodgepodge of suspense, scenery, and intrigue, good traps can have whole stories behind them, and can combine the best of timing/sequence and mental deduction puzzles.

Quests

Another type of common game challenge, mentioned in Chapter 2, "A Brief Look at 2D Games," is the quest. According to Jeff Howard, author of *Quests:*

Figure 7.3
Quest design can go for fairy tales as well as popular RPGs.

Design, Theory, and History in Games and Narratives, published by A.K. Peters, "a quest is a journey across a symbolic, fantastic landscape in which a protagonist or player collects objects and talks to characters in order to overcome challenges and achieve a meaningful goal." This could describe *Alice in Wonderland* or *Elder Scrolls IV: Oblivion*, couldn't it? See Figure 7.3.

Quests are further broken down into their challenge subtypes:

- **Fetch**: the player must find a valuable, often magically important object, and either use it at an opportune time or return it to someone who needs it. There are also occasional FedEx-type quests, where the player has to deliver an important object from one non-player character (NPC) to another.
- **Combat**: requires the player to slay monsters or hostile NPCs, either as an end in itself to remedy an injustice or as a means to an end, such as in an ever-popular dungeon crawl.
- **Escort/Chase**: an infrequently used quest, but one with built-in suspense, is where the player must guide an NPC from one location to another while making sure no harm befalls them—also called the "escort quest." Similarly, another type of quest is where the player must chase and try to do harm to a hostile NPC while that NPC is attempting to get away—also called the "chase quest."

Don't Forget to Add Conflict

You can imagine any game challenge you've witnessed before in another game, or perhaps invent a new one. One of the key ingredients of any challenge that keeps players involved is conflict.

Chris Crawford once said, "Conflict is an intrinsic element of all games. It can be direct or indirect, violent or nonviolent, but it is always present in every game." He went on to describe why: "Conflict implies danger, danger means risk of harm, and harm is undesirable." Every living thing is afraid of harm; no one likes to get hurt! Even human beings who are long-time sufferers, the so-called victims of learned helplessness, are not all that keen on being hurt; they get used to greater rewards coming from being hurt.

The element of conflict is simple. In gambling halls and casinos, the method by which people are hurt is that they lose money at the craps tables and slot machines. It is the conflict that drives them to take the chance and win big or lose it all.

The same is true for video games. In fact, a few years ago there was a small article that came out that said video games were similar to an addictive substance, because they have all the telltale signs of a bad habit. I'm not saying that video games turn gamers into gamblers. There's no direct correlation between gambling (or even drug usage) and games. However, games *have* been known to be just as addictive as hitting casinos or doing drugs, so if a person with natural addictive personality traits discovers a game he likes, he could become just as addicted to it as if it were a bad gambling habit or drug!

Most video games feature "hit points" where—after getting hit enough times—the player's onscreen avatar will expire. This is why so many video games are slammed by the media for being too violent.

When asked if a video game could be non-violent, veteran game designer Warren Spector (Figure 7.4), the creator of games like *Thief: Deadly Shadows* and *Deus Ex: Invisible War*, said that it was frankly impossible. Conflict implies the risk of winning or losing, and in games it is often how many times you get fragged that decides the conflict!

One Lesson to Remember

Keep one of the Four Fs of Great Game Design in mind when designing game challenges: Fairness.

Figure 7.4
An industry icon and master speaker at game conventions, Warren Spector has gems of advice to give newbie developers.

Don't let an otherwise great game bottleneck because you put in a monster that is nearly unbeatable or a cryptogram only a science professor could decipher. Along this same line of thought, avoid creating arbitrary challenges or challenges only you, the creator, would know how to overcome. If players get stuck, they will turn to an online walkthrough guide, and if they have to resort to reading an online walkthrough guide, you have done a poor job of designing your game!

Never force a player to learn by dying. Always help the player to figure something out before slapping their wrists. They'll thank you for it. This should be a game, after all, not work!

Game challenges should *never* be monotonous. For instance, if the player has to get a red key to open a red door, a green key to open a green door, and a blue key to open a blue door, just to find a sword that turns out be just as efficient as the one the player already owns, then that player is going to shoot your game down by poor word-of-mouth advertisement, and there's no blaming him!

So be fair when crafting your game and you'll find that more players thank you for it.

Creating Game Challenges in TGB

Tip

> "First act, get your leading character up a tree; second act, throw rocks at him; third act, get him down."
>
> —Allen Saunders

Now we can set up a challenge in the game project we started in Torque Game Builder. First we must set up the reward, or the promise of reward, and then we must design obstacles between the player and that reward. This will create conflict, and we will have accomplished our purpose of creating a game with Torque Game Builder.

Loading Your Resources

Before continuing with this exercise, you will need to copy the files you need from the data files on the companion CD that comes with this book and place them into your work folder.

1. Momentarily leaving the Torque Game Builder Level Editor, go to the Chapter 7 folder in the data files found on the CD-ROM.
2. Select both the image files inside the Chapter 7 folder and Edit > Copy (shortcut Ctrl + C for Win users, CMD + C for Mac users) to copy them to your Clipboard.
3. Browse to your MyCastle\Game\Data\Images folder and paste the image files from your Clipboard into it by using Edit > Paste (shortcut Ctrl + V for Win users, CMD + V for Mac users).

Adding a Reward

The player's main goal will be to move Batty around and gather coins that fall from the sky. Later, we will add hindrances that impede Batty from swiftly getting them all, but for now we set the reward system up and make sure it works.

Programming the Reward

We are going to create a separate program file for our reward, but to do that we will have to call it from within the main game.cs file, like we did for player.cs.

Open your game.cs file in your default text editor. Modify the `startGame` function to look like this:

```
function startGame(%level)
{
  exec("./player.cs");
  exec("./reward.cs");

  Canvas.setContent(mainScreenGui);
  Canvas.setCursor(DefaultCursor);

  new ActionMap(moveMap);
  moveMap.push();

  $enableDirectInput = true;
  activateDirectInput();
  enableJoystick();

  sceneWindow2D.loadLevel(%level);
}
```

Save game.cs and create a new text file within the same folder and call this new text file reward.cs, making sure to add the .cs extension so the game engine recognizes it as program code for use with Torque.

Open reward.cs in your text editor and enter this bit of code:

```
function BigReward::onLevelLoaded(%this, %scenegraph)
{
  %this.startPosition = %this.getPosition();
  %this.setLinearVelocityY(getRandom(%this.minSpeed, %this.maxSpeed));
}
```

This function is pretty simple. We store the start position of the object and then set its linear velocity along the Y axis, which is vertical up and down the screen. We get a random number somewhere between its `minSpeed` and `maxSpeed`. These are variables we will have to specify in the Level Builder later.

For now, our code looks good, but it won't work. If the coin continues falling along the Y axis, and the player isn't quick enough to grab it, it will fall right out of the level and won't return. The game would be awfully short and frustrating. We need to return the coin back up to the top if it falls off the bottom of the screen. We do that with the world limit. Type this function after your last one:

```
function BigReward::onWorldLimit(%this, %mode, %limit)
{
  if(%limit $= "bottom")
   {
   %this.spawn();
   }
}
```

If it is indeed the bottom edge of the world limit, then the `spawn()` function is called. Now let's add the `spawn()` function:

```
function BigReward::spawn(%this)
{
  %this.setPosition(getRandom(-50,50), %this.startPositionY);
  %this.setLinearVelocityY(getRandom(%this.minSpeed, %this.maxSpeed));
}
```

Notice how similar this function is to `onLevelLoaded()`? The only difference is that it sets the coin's position to the start position stored when the object was first loaded into the level, but at a random X position left or right 50 pixels. It then proceeds to have the coin fall again, taking with it a random speed value. This way, each time it spawns it will have a new start position and new speed, making it look like an entirely different coin.

Save your reward.cs file and return to the Level Builder.

Creating the Reward

Once you load the Level Builder, you should be greeted with your level and your Batty guy in the midst of the castle scene. First thing we need to do is bring the visual representation of our reward, a coin image, into our level.

1. Go to the Create tab on the right-hand side of the editor.
2. Click the Create a New Image Map button under the tab.
3. Locate the file coin.png. Click OK and the coin will become a sprite in the Static Sprites section.
4. Double-click on the coin thumbnail image under Static Sprites to open the Image Builder. We want to animate this coin just like we did Batty and the flames, so change the coin's Image Mode to CELL, as you see in Figure 7.5. Click Save when you're through.

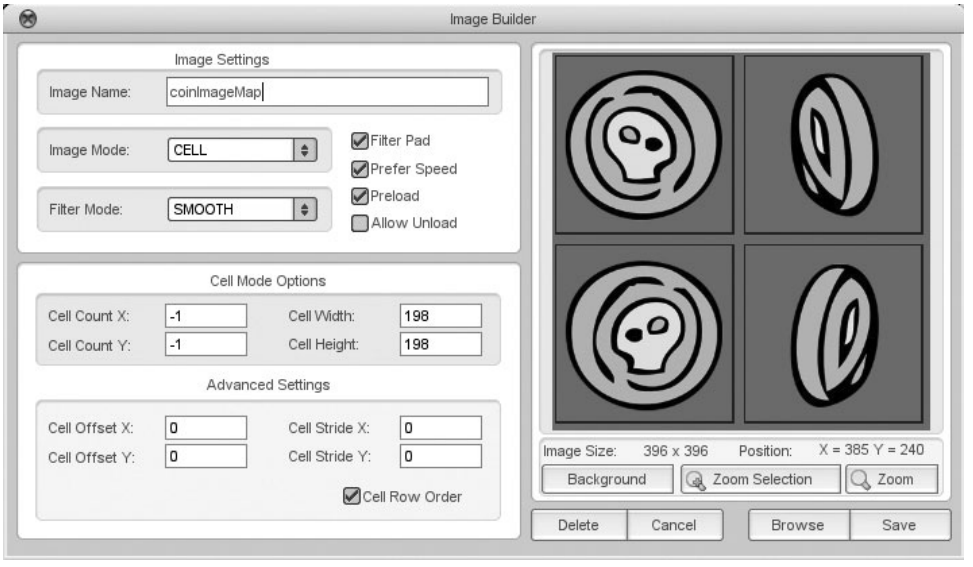


Figure 7.5
Set the coin's Image Mode to CELL.

5. Click the Create a New Animation button under the Create tab and select `coinImageMap` as your source. In the Animation Builder, click the Add all frames from the image green plus sign button and change the Frames Per Second to a value of 4 before clicking the Save button.

6. Under the Create tab, go to the Animated Sprites section, and drag-and-drop your coin image into your level on the left, resizing it and placing it just above your camera view (see Figure 7.6).

7. With the coin image still selected, roll over it with your mouse cursor. You will see a set of icons appear above the image. Click the second icon from the right, which is the World Limit tool.

8. Scale the world limits box so that the left and right sides are just a little ways outside the camera view, because we want the coin to appear offscreen from the top and disappear offscreen at the bottom. Compare your work to Figure 7.7.

9. Once you've set the world limit boundary, click the Selection tool in the top toolbar to exit the world limit editing mode.

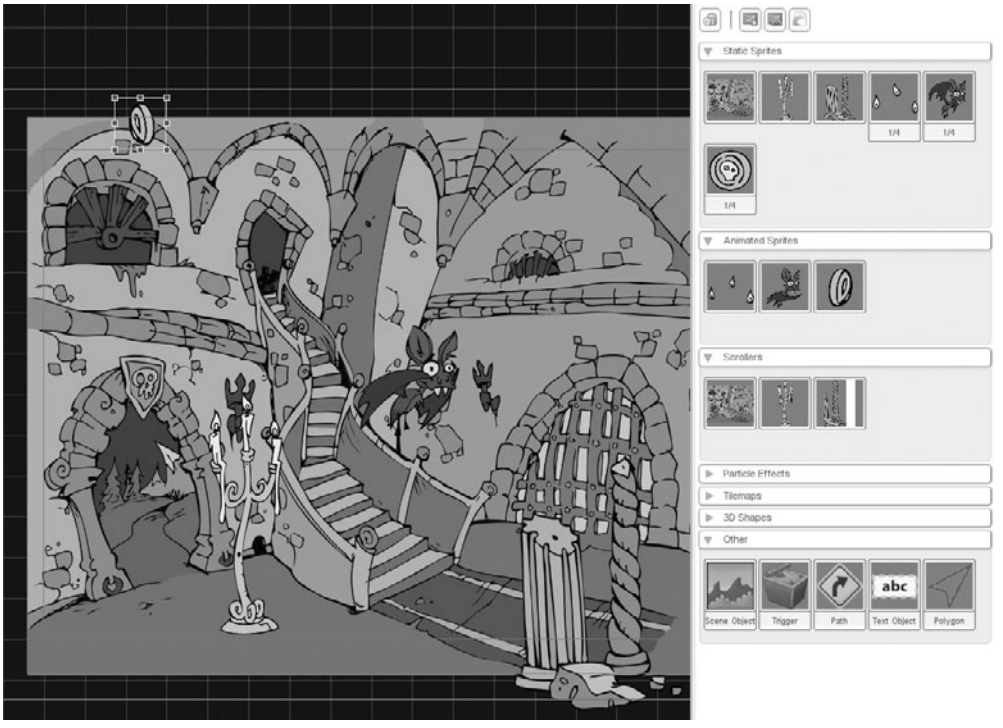


Figure 7.6
Set the coin image in your castle environment, just above the top edge of the camera view.

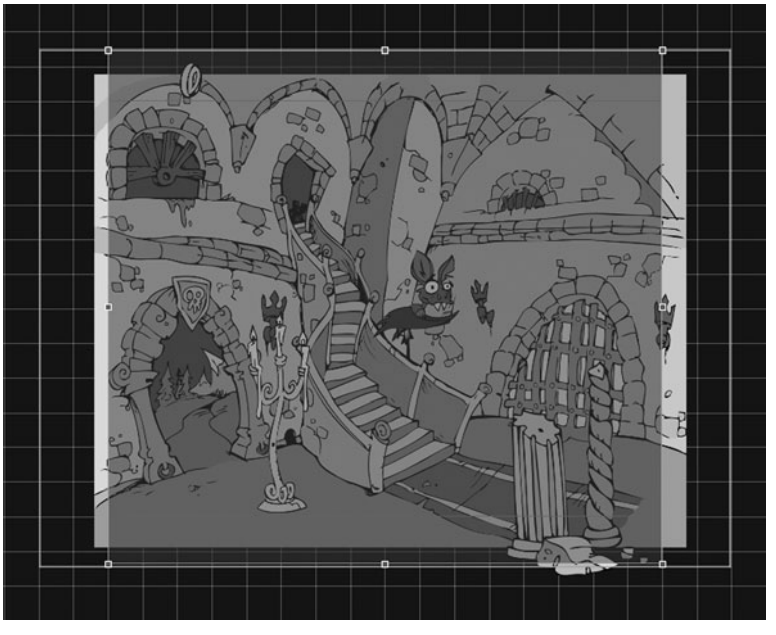


Figure 7.7
Scale the coin's world limits to fit just outside the camera view.

Figure 7.8
Create the two variables `minSpeed` and `maxSpeed` and set their values accordingly.

10. Click the Edit tab and go to the Scene Object to find the Layer property. You want to set the coin's Layer to 10, before going to the World Limits rollout. The default is set to OFF but you want to change it to NULL. There are plenty of prefabricated responses, but NULL should allow us to use our program function to do what we want with the coin.

11. With the coin still selected, go to the Dynamic Fields rollout under the Edit tab. We want to create two variables, `minSpeed` and `maxSpeed`. Type **`minSpeed`** in the Field Name section and **5** for the Field Value, then press the little green plus sign to add the field. Do the same for **`maxSpeed`**, setting its Field Value to 15. Compare your work to Figure 7.8.

12. Go to the Scripting rollout under the Edit tab and type **`BigReward`** in the Class input field and hit Enter.

We are done setting up our reward system. Now it's time to test it. Save your level and click the Play Level button. You should see the coin slowly fall through the level, then pass out of view, only to appear back at the top with a different start position and random speed. However, it doesn't actually *do* anything else . . . and our Batty character can't grab any of the shiny coins.

Giving the Coins to Batty

We want to script it so that when Batty touches a coin, the coin disappears, gives Batty a boost, and then respawns back at its start position. To do this, we have to utilize the collision system native to Torque Game Builder.

Editing Collision Polygons First, we will enable collision on our coins and Batty, then set up proper collision polygons on both. A *collision polygon* is an

invisible but very important border outlining an object that the game engine registers as the boundary for collision responses.

1. In the Level Builder, select the coin image.
2. Click on the Collision rollout bar (found under the Edit tab) to expand your collision options.
3. Check Send Collision and Callback, and uncheck Receive Collision, Send Physics, and Receive Physics, as seen in Figure 7.9.

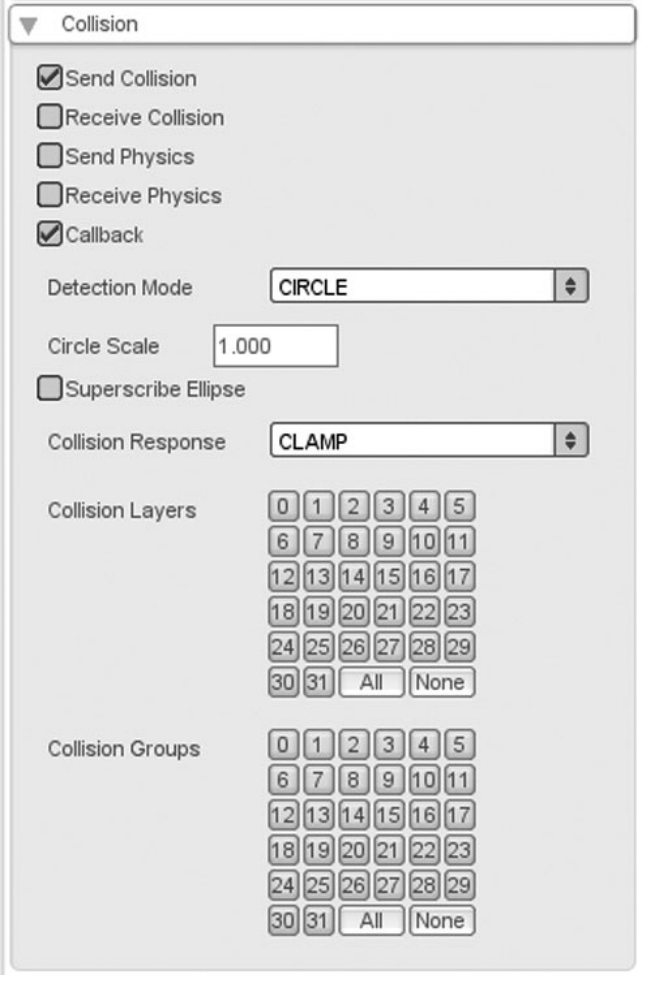


Figure 7.9
Be sure to set the collisions correctly.

4. Click the Detection Mode drop-down and select CIRCLE. This automatically provides you with a circular collision polygon.

5. Once you have chosen CIRCLE, uncheck Superscribe Ellipse, as this would hamper our detection mode for the coin.

6. Select your Batty character in the scene and hover over it with your mouse cursor. You will see a set of icons appear above the image. Click the first icon button from the left, which is the Edit Collision Polygon button (see Figure 7.10), to enter the Collision Polygon editing mode.

7. Your view should zoom in until your Batty character is the only thing you should see. Once in this mode, it is time to define the object's custom collision polygon.

8. There are already four default anchor points for your collision polygon, which you can click-and-drag into place where you want them. Wherever

Figure 7.10
The Edit Collision Polygon icon button is the first over from the left.

you click your mouse, a new anchor point for your collision polygon will be created. Go around your object, tracing a boundary outlined between anchor points (as seen in Figure 7.11). It's a good idea to keep your collision polygon simple, say between five and ten anchor points long. Note that you *cannot* create concave collision polygons, so if you find that your anchor point is not moving where you want it to go, it is probably because you are trying to make a concave shape, and the Level Builder won't let you do that.

9. Once you've set the collision polygon, click Save to exit the Collision Polygon editing mode.

10. Go to the Edit tab and find the Collision rollout there.

11. Check Send Collision and Callback, and uncheck Receive Collision, Send Physics, and Receive Physics. Ignore the Detection mode, because it is already set to POLYGON, and we've just designed a custom polygon.

12. Save your level.

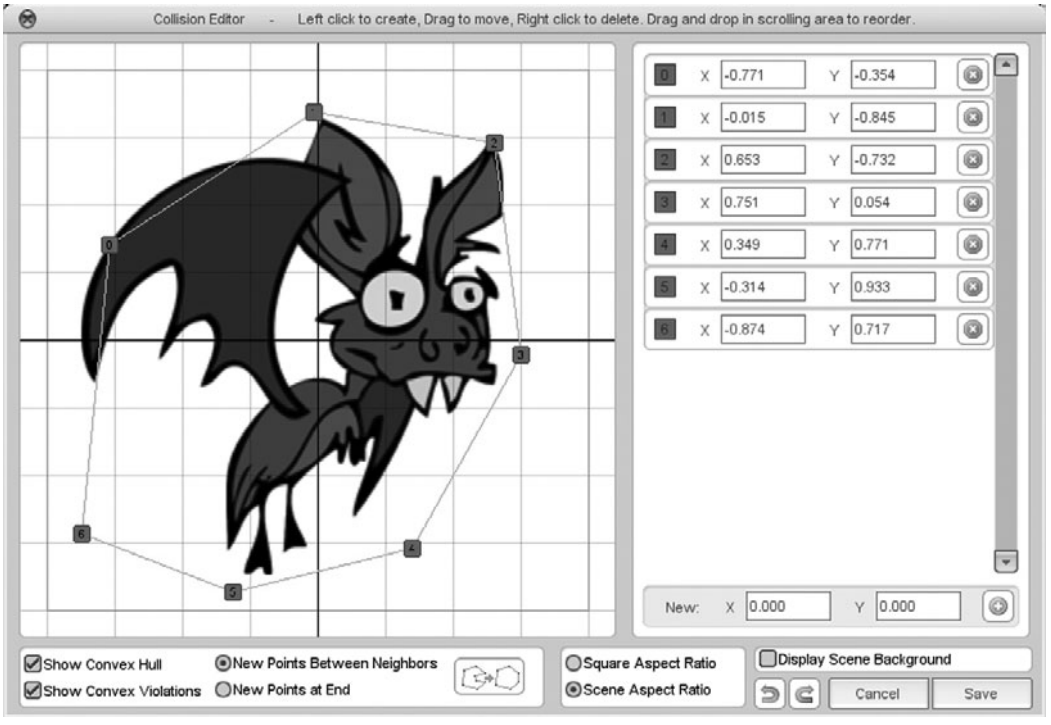


Figure 7.11
Trace a collision polygon around Batty by clicking to add anchor points.

Programming Collision Responses Open your reward.cs file in your default text editor. Above the `BigReward::spawn()` function but below the `BigReward::onWorldLimit()` function, type in this code:

```
function BigReward::onCollision(%srcObj, %dstObj, %srcRef, %dstRef, %time,
%normal, %contactCount, %contacts)
{
  if(%dstObj.class $= "BattyPlayer")
   {
    %srcObj.spawn();
   }
}
```

Collisions pass back all kinds of important information, but the ones we are using here are `%dstObj` and `%srcObj`. Our `%srcObj` in this case is our coin, and our `%dstObj` is our Batty character. The condition checks to see if the coin has come in contact with the `BattyPlayer` object, and if so, it tells it to `spawn()` the coin.

Your reward.cs file should now read like this:

```
function BigReward::onLevelLoaded(%this, %scenegraph)
{
  %this.startPosition = %this.getPosition();
  %this.setLinearVelocityY(getRandom(%this.minSpeed, %this.maxSpeed));
}

function BigReward::onWorldLimit(%this, %mode, %limit)
{
  if(%limit $= "bottom")
   {
   %this.spawn();
   }
}

function BigReward::onCollision(%srcObj, %dstObj, %srcRef, %dstRef, %time,
%normal, %contactCount, %contacts)
{
  if(%dstObj.class $= "BattyPlayer")
   {
    %srcObj.spawn();
   }
}
```

```
function BigReward::spawn(%this)
{
  %this.setPosition(getRandom(-50,50), %this.startPositionY);
  %this.setLinearVelocityY(getRandom(%this.minSpeed, %this.maxSpeed));
}
```

Providing Competition

Just because the player can move Batty around to grab coins, that is not enough. Now you have to add an obstacle. In this, you create completion. Have other bats rush for the coins, too. To set up some other bats, you first need to create the bats and then program them to go after the coins.

Creating Other Bats

The bats that we will add to our level should not look like Batty, because the player should not get confused or wonder which character is his avatar. The bat created is similar but smaller and different-colored than Batty, to make it clear that it is not Batty.

1. Go to the Create tab and click the Create a New Image Map button under the tab.

2. Locate the bat.png file. Simply click OK and the bat will become a sprite in the Static Sprites section of your editor, under the Create tab.

3. Double-click on the bat thumbnail image under Static Sprites to open the Image Builder. We want to animate this bat, so change the bat's Image Mode to CELL. Click Save when you're through.

4. Click the Create a New Animation button under the Create tab and select `batImageMap` as your source. In the Animation Builder, click the Add all frames from the image green plus sign button and change the Frames Per Second to a value of 10 before clicking Save.

5. Go to the Animated Sprites section and drag-and-drop your bat image into your level on the left, resizing it and placing it on the right just outside of your camera view (see Figure 7.12).

6. With the bat image still selected, roll over it with your mouse cursor and click the World Limit Tool icon button.

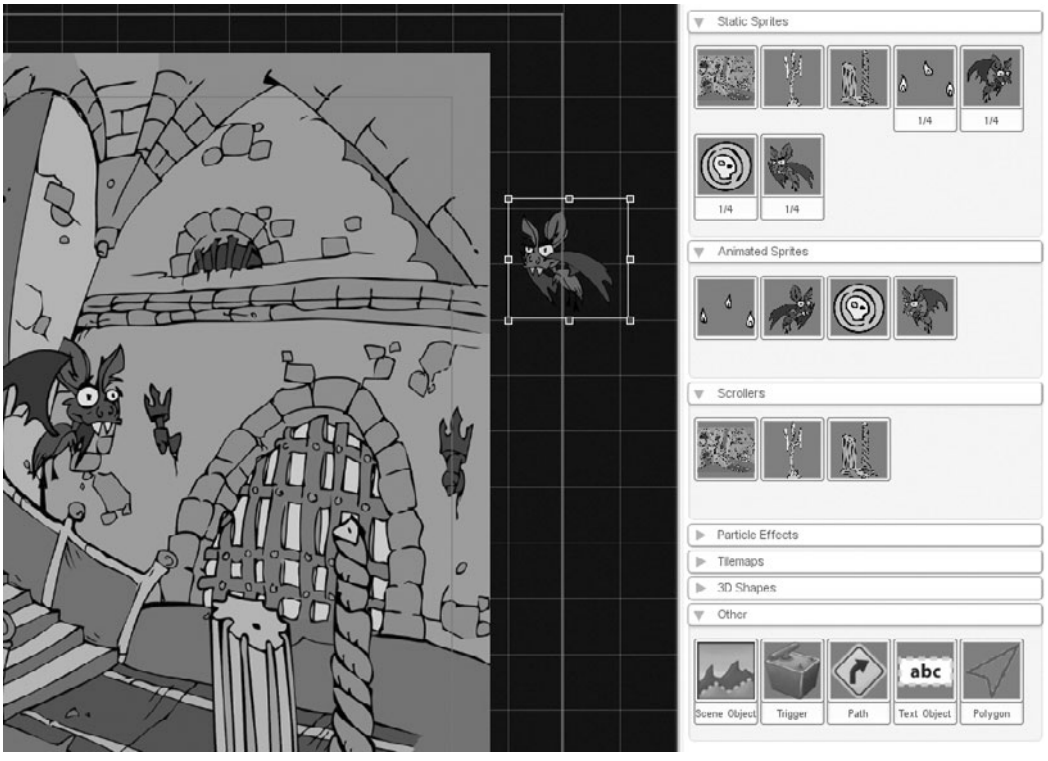


Figure 7.12
Set your enemy bat into your castle environment.

7. Scale the world limit box so that the left and right sides are well outside the camera view, because we want the bat to disappear offscreen before it returns.

8. Once you've set the world limit boundary, click the Selection tool in the top toolbar to exit the World Limit editing mode.

9. Click the Edit tab and go to the Scene object to find the Layer property. You want to set the coin's Layer to 10, before going to the World Limits rollout. The default is set to OFF, but you want to change it to NULL. There are plenty of prefabricated responses, but NULL should allow us to use our program function to do what we want with the bat.

10. With the bat still selected, go to the Dynamic Fields rollout under the Edit tab. We want to create two variables, `minSpeed` and `maxSpeed`. Type **`minSpeed`** in the Field Name section and **5** for the Field Value, then press

the little green plus sign to add the field. Do the same for `maxSpeed`, setting its Field Value to "**30**".

11. Hover over the bat with your mouse cursor. Click the first icon button from the left, the Edit Collision Polygon button, to enter the Collision Polygon editing mode.
12. Once in this mode, it is time to define the object's custom collision polygon.
13. There are four default anchor points, and wherever you click your mouse a new anchor point for your collision polygon is created. Go around your object, tracing a boundary outlined between anchor points.
14. Once you've set the collision polygon, click Save to exit Collision Polygon editing mode.
15. Go to the Edit tab and scroll down to the Collision rollout.
16. Check Send Collision and Callback, and uncheck Receive Collision, Send Physics, and Receive Physics. Ignore the Detection mode, because it is already set to POLYGON, and we've just designed a custom polygon.
17. Go to the Scripting rollout under the Edit tab and type **`MeanBat`** in the `Class` input field and press Enter.

To make it look like Batty is competing against several bats, select the bat image you have just made in the Level Builder and press Ctrl + C (Win) or CMD + C (Mac) to copy the ghost instance to the Clipboard. Then press Ctrl + V (Win) or CMD + V (Mac) to paste the bat instance from the Clipboard. Click and drag the new bat to another location or use the up and down arrow keys to nudge them one way or the other, as seen in Figure 7.13.

Repeat until you have as many opponents as you desire, depending on the difficulty level.

Programming the Bats

Just as we created a player script and a reward script, we will create a separate script file for the bats. In the main game.cs file, edit the `startGame` function to read as such:

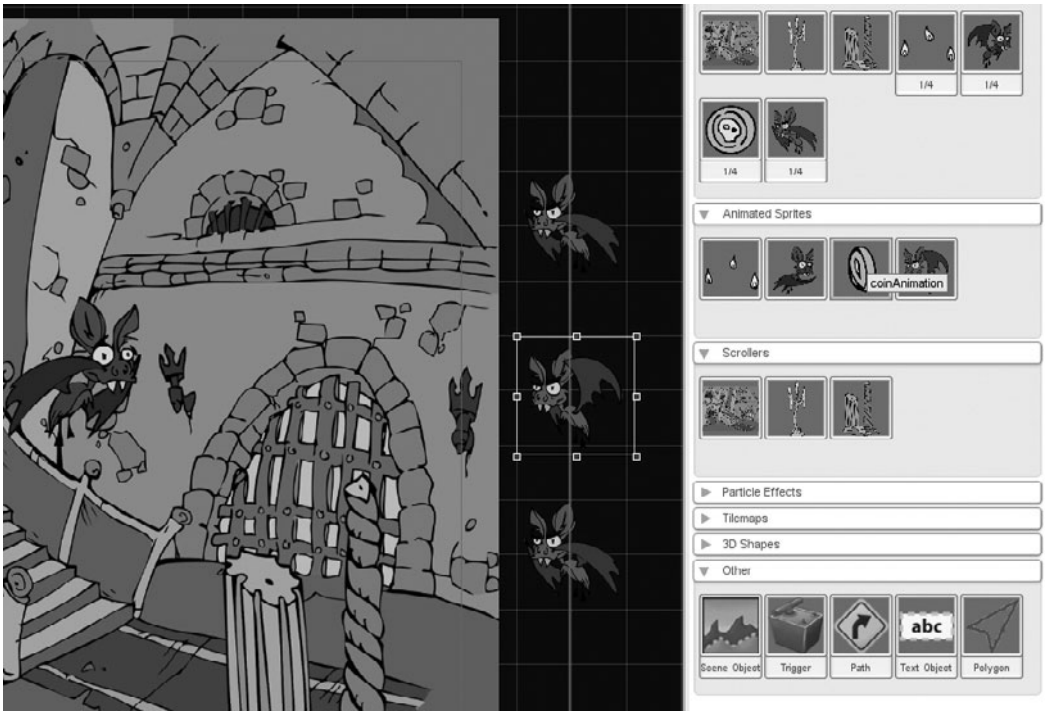


Figure 7.13
Change the frame number of your duplicate bat to make it appear different.

```
function startGame(%level)
{
  exec("./player.cs");
  exec("./reward.cs");
  exec("./enemy.cs");

  Canvas.setContent(mainScreenGui);
  Canvas.setCursor(DefaultCursor);

  new ActionMap(moveMap);
  moveMap.push();

  $enableDirectInput = true;
  activateDirectInput();
  enableJoystick();

  sceneWindow2D.loadLevel(%level);
}
```

Save the game.cs file. Create a new text file in the same folder as the rest and call it enemy.cs. Open enemy.cs in your text editor.

The bat will behave much like the coin, but instead of falling from the top to the bottom of the camera view, we want it to move left and right, randomly reposition itself, and whenever it crosses a coin, we want the bat to make the coin disappear. Let's start by coding the `onLevelLoaded()` function. Add the following to your enemy.cs file:

```
function MeanBat::onLevelLoaded(%this, %scenegraph)
{
  %this.setLinearVelocityX(-%this.getRandom(%this.minSpeed, %this.maxSpeed));
}
```

This script loads the bat and tells it to move at a random velocity, somewhere between `minSpeed` and `maxSpeed`, the two variables we set up in the Level Builder, along the X axis horizontally across the screen.

Next we need to add a function for when the bat reaches its left or right world limits:

```
function MeanBat::onWorldLimit(%this, %mode, %limit)
{
  switch$(%limit)
  {
   case "left"
   %this.setFlipX(true);
   %this.setLinearVelocityX(%this.getRandom(%this.minSpeed, %this.maxSpeed));
   %this.setPositionY(getRandom(-35, 35));

   case "right"
   %this.setFlipX(false);
   %this.setLinearVelocityX(-%this.getRandom(%this.minSpeed, %this.maxSpeed));
   %this.setPositionY(getRandom(-35, 35));
  }
}
```

This is fairly straightforward. When the bat reaches the left edge of its world limits, it will be flipped over to face the other direction. Originally, the bat image is facing left, and it will start by flying left. So when it hits the left of the screen, it must be flipped over to face the opposite direction. This is accomplished by

returning `setFlipX(true)`. Then it starts flying again at a random velocity between `minSpeed` and `maxSpeed`, coming in from a random position along the Y axis (up or down approximately 35 pixels).

When the bat reaches the right edge of its world limits, it will be flipped over to face the left again, and since that is its default look, we do this by returning `setFlipX(false)`. Then it starts flying again, once more coming in at a random position up or down the screen about 35 pixels.

The last thing to do before we save our program script is to add a snippet for collision with the coins. Open up your reward.cs file briefly and scroll to where you added the `BigReward::onCollision` code. Alter it to look like this:

```
function BigReward::onCollision(%srcObj, %dstObj, %srcRef, %dstRef, %time,
%normal, %contactCount, %contacts)
{
  if(%dstObj.class $= "BattyPlayer" || %dstObj.class $= "MeanBat")
   {
    %srcObj.spawn();
   }
}
```

The symbol "||" in code talk means "or"—so what we're telling the game engine is that if the object the coin is colliding with is the player character (`BattyPlayer`) or the player's opponent (`MeanBat`), then it should instantly respawn.

There you go! Save your reward.cs file and your enemy.cs file before continuing.

Proofreading Your Program

Your reward.cs file should look like this:

```
function BigReward::onLevelLoaded(%this, %scenegraph)
{
  %this.startPosition = %this.getPosition();
  %this.setLinearVelocityY(getRandom(%this.minSpeed, %this.maxSpeed));
}

function BigReward::onWorldLimit(%this, %mode, %limit)
{
  if(%limit $= "bottom")
   {
   %this.spawn();
```

```
    }
}
function BigReward::onCollision(%srcObj, %dstObj, %srcRef, %dstRef, %time,
%normal, %contactCount, %contacts)
{
  if(%dstObj.class $= "BattyPlayer")
   {
    %srcObj.spawn();
   }
}

function BigReward::spawn(%this)
{
  %this.setPosition(getRandom(-50,50), %this.startPositionY);
  %this.setLinearVelocityY(getRandom(%this.minSpeed, %this.maxSpeed));
}
```

And your enemy.cs file should look like this:

```
function MeanBat::onLevelLoaded(%this, %scenegraph)
{
  %this.setLinearVelocityX(-%this.getRandom(%this.minSpeed, %this.maxSpeed));
}

function MeanBat::onWorldLimit(%this, %mode, %limit)
{
  switch$(%limit)
  {
   case "left"
   %this.setFlipX(true);



   %this.setLinearVelocityX(%this.getRandom(%this.minSpeed, %this.maxSpeed));
   %this.setPositionY(getRandom(-35, 35));

   case "right"
   %this.setFlipX(false);
   %this.setLinearVelocityX(-%this.getRandom(%this.minSpeed, %this.maxSpeed));
   %this.setPositionY(getRandom(-35, 35));
  }
}
```

If you find any errors, change them now or refer back to this section when you encounter errors during play-testing.

Testing Your Level

Open the Level Builder. Save your level and click the Play Scene button. Watch, as it has become more difficult to grab coins, because now there are a bunch of nasty bats to race against to get the falling treasure, as seen in Figure 7.14! If the difficulty is too hard, and you find you are having too much trouble reaching the coins before the other bats, then you might decrease the number of bats you have to compete against in the Level Builder. If one of your objects—your player character, the other bats, or the coins—aren't behaving right, or if you have cascading errors, you might open up your game.cs file and double-check your code, because it's very easy to misspell something or forget and leave out a semicolon or curly bracket.

Figure 7.14
The finished game project.

Once you have your first game project finished, it's time to expand your horizons and begin a new project. That's right . . . This time you will program a basic shooter game!

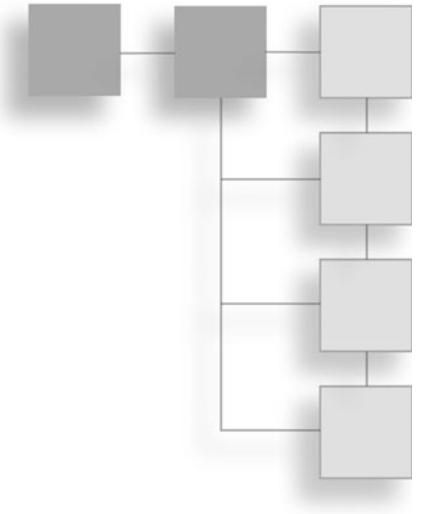
Review

After reading this chapter, you should understand the following:

- The most common game challenges
- Why it's important to add conflict in games but remain fair to your players
- How to add rewards to your games and use the TGB collision system
- How to add obstacles or opponents to your games

CHAPTER 8

Making a Shooter



In this chapter, you will learn:

- To add an enemy that moves across the screen
- To spawn enemies at random positions
- To make the enemies destroy the player if they touch him
- To give the player a weapon to shoot the enemies with
- To add background music and weapon-fire sound effects

Now that you've finished making a playable game of your own, why not branch out and try a very popular game genre, the shooter? A shooter, if you recall, is an action game that focuses mainly on the hand-eye coordination and fast reflexes of the player, involving shooting at enemies and blowing stuff up. Sure, it's adolescent in its culture, but the shooter game is one of the most widely recognized and copied game genres in existence.

This chapter will show you everything you need to turn your Batty into a vicious air assault weapon!

Characteristics of the Shooter

Shooter games are a subgenre of the action game genre. Gameplay emphasis is fast action and twitch reflexes, mostly involving shooting things. Because shooters make up the majority of action games, it is a fairly wide subgenre.

Shooters involve an avatar, one or more ranged weapons, and a varying number of enemies.

In a shooter, the player usually views the situation from behind the eyes of the character, as in the first-person shooter, or from a camera that follows the character. Most shooters involve varying levels of realism, with some verging on complete fantasy. While most shooters are played as solo ventures, others offer players the opportunity to control a whole squad of characters and give orders to computer-controlled allies.

Online shooters have different rules, as players can assign teams, play co-op (cooperating together long-distance to reach the same goals), or play deathmatch competitions against one another.

Loading a Saved Project

We don't want to use the catch-the-coins game project for this next exercise, but we don't want to have to go back and redo everything in Chapter 6. There is a saved game project on the companion CD for this book that has everything you need to get started on your shooter game.

1. Momentarily leaving the Torque Game Builder Level Editor, go to the Chapter 8 folder in the data files found on the CD-ROM.

2. Select the Project subfolder inside the Chapter 8 folder and Edit > Copy (shortcut Ctrl + C for Win users, CMD + C for Mac users) to copy it to your Clipboard.

3. Browse to your work folder, where you've been saving your projects so far, and in the upper hierarchy, paste the Project folder from your Clipboard into it by using Edit > Paste (shortcut Ctrl + V for Win users, CMD + C for Mac users).

4. In the TGB Level Builder select File > Open Project. Browse to the Project folder and select project.t2dProj to open it. You're ready to begin!

Creating Enemies to Fight

We already have our player character, Batty, completed for us. Batty can't attack just yet, which we'll get around to fixing, but for now let's focus on giving him an opponent to worry about.

Creating a Ghost Enemy

Since we are in a big spooky castle, playing a flying mammal, we should have a supernatural enemy to defeat. How about a ghost?

1. Under the Create tab on the right-hand side of the editor, click the Create a New Image Map button under the tab.
2. Locate the ghost.png file. Click OK and the ghost will become a sprite in the Static Sprites section, under the Create tab.
3. Double-click on the ghost thumbnail image under Static Sprites to open the Image Builder. Change the ghost's Image Mode to CELL. Click Save when you're through.
4. Click the Create a New Animation button under the Create tab and select `ghostImageMap` as your source. In the Animation Builder, click the Add all frames from the image green plus sign button and change the Frames Per Second to a value of 10 and check the Random box before clicking the Save button.
5. Under the Create tab, go to the Animated Sprites section and drag-and-drop the enemy sprite into the scene and resize it to fit to the perspective of the environment, as shown in Figure 8.1.
6. Under the Edit tab, change the ghost's Layer object property to 10.
7. Now that we have the enemy in our scene, let's get him moving across the screen. Just like the `BattyClass`, we will have to give our enemy a name and a class. Click the Edit tab, with your ghost selected, and type in a class of `enemyGhost`.

Programming the Ghost Enemy

Open your game.cs file from the Project\Game\GameScripts folder in your default text editor.

Modify it as such:

```
//-----------------------------------------------------------------------------
-----------------
// Torque Game Builder
// Copyright (C) GarageGames.com, Inc.
//-----------------------------------------------------------------------------
-----------------
```

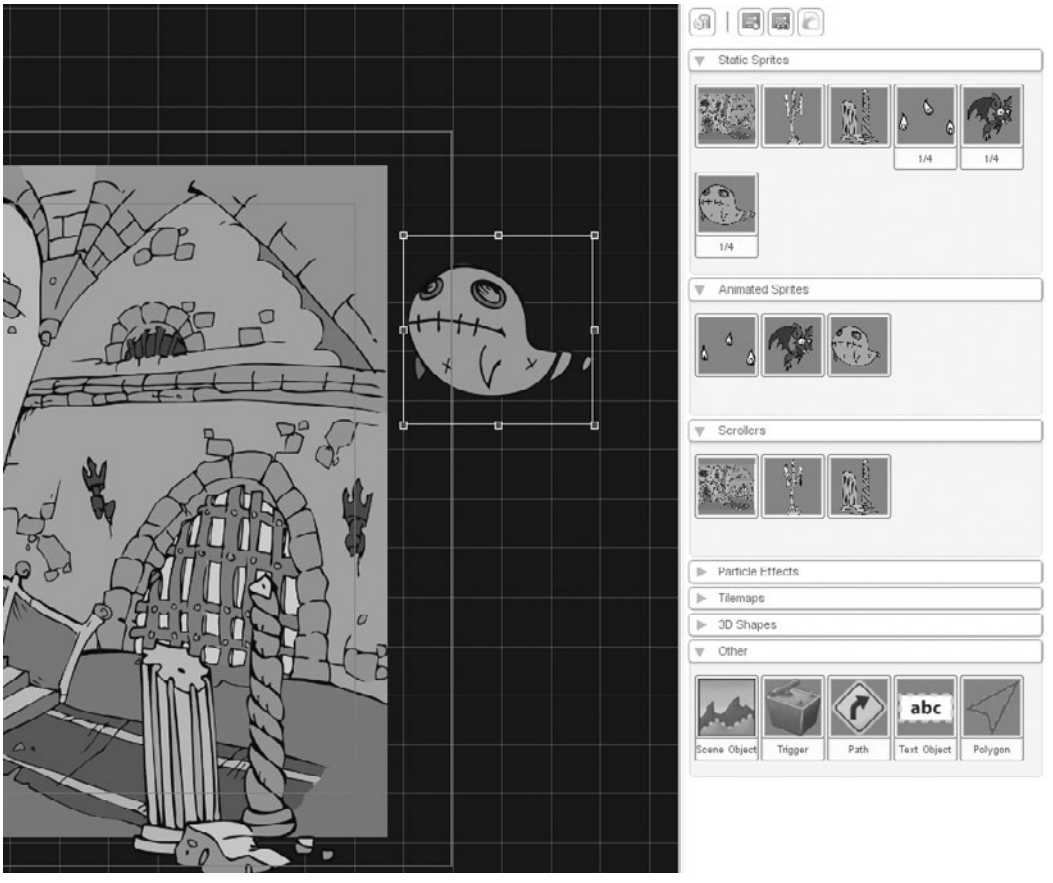


Figure 8.1
Set up the ghost image in your castle scene.

```
//--------------------------------------------------------------------------
------------------
// startGame
// All game logic should be set up here. This will be called by the level builder
when you
// select "Run Game" or by the startup process of your game to load the first level.
//--------------------------------------------------------------------------
------------------
function startGame(%level)
{
  exec("./player.cs");
  exec("./enemy.cs");
```

```
  Canvas.setContent(mainScreenGui);
  Canvas.setCursor(DefaultCursor);

  new ActionMap(moveMap);
  moveMap.push();

  $enableDirectInput = true;
  activateDirectInput();
  enableJoystick();

  sceneWindow2D.loadLevel(%level);
}

//--------------------------------------------------------------------------
------------------
// endGame
// Game cleanup should be done here.
//--------------------------------------------------------------------------
------------------
function endGame()
{
  sceneWindow2D.endLevel();
  moveMap.pop();
  moveMap.delete();
}
```

Just as we did before, but with a new intent, we're going to create a separate script file to contain all our enemy code, and we'll call it enemy.cs. In the same folder where you found the game.cs file, create a new text file and call it enemy.cs. In your text editor, type the following code into enemy.cs:

```
function enemyGhost::onLevelLoaded(%this, %scenegraph)
{
  %this.enemyMovement();
}
function enemyGhost::enemyMovement(%this)
{
  %this.setLinearVelocityX(getRandom(%this.minSpeed, %this.maxSpeed));
}
```

The first function calls the second, which sets its speed along the X axis at a random variable somewhere between `minSpeed` and `maxSpeed`. These are variables

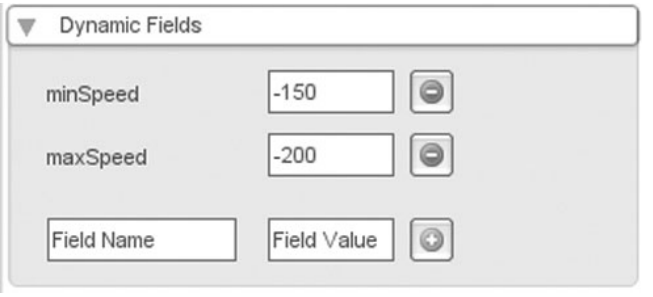


Figure 8.2
Enter a `minSpeed` and `maxSpeed` variable in the Dynamic Fields rollout.

that don't exist yet, so we will have to add them to our ghost object, just like we did the `Batty` object.

In the Level Builder, select the enemy object and go to the Edit tab. Scroll down to find the Dynamic Fields rollout. Add a `minSpeed` and `maxSpeed` variable, as you see in Figure 8.2. You can set these variables to anything you desire, depending on how difficult you want your enemy to be, keeping in mind that `minSpeed` must be a lower value than `maxSpeed`. One suggestion is to try them at -150 and -200, respectively, and then adjust to your liking. The numbers are negative, because the enemy will be moving in the opposite direction as the player character. Once you have your random speed range set, save your level and play test it.

Setting the Ghost's World Limits

Notice that when the ghost gets to the edge of the screen, he disappears completely, never to reappear. This is boring. We want the ghost to come back as soon as he hits the edge of the screen. So we're going to have to come up with a script to make him respawn as soon as he hits his world limits.

But first we must define his world limits, just as we did the player's in Chapter 6.

1. In the Level Builder, select the ghost image and roll over it with your mouse cursor. You will see a set of icons appear above the image. Click the second icon from the right, which is the World Limit tool.

2. Your view will change until it is zoomed way out from your scene. Plus, you should see a scalable box representing the world limit of your entity. At first the limit far exceeds the scene. Resize the ghost's world limits to fit just outside the camera view, such as in Figure 8.3. You want to leave enough

Figure 8.3
Extend the world limits boundary past the camera view so that the enemy can disappear out of view before reappearing from the right.

room so that when the ghost wanders outside camera view and is completely gone from the player's view, he can be blipped back to the other side and respawned out of view.

3. Once you've set the world limit boundary, click the Selection tool in the top toolbar to exit the World Limit editing mode.

4. Click the Edit tab and scroll down to find the World Limits rollout. The default is set to OFF but you want to change it to NULL. There are plenty of prefabricated responses, but NULL will allow you to add a scripted response.

5. Click the callback checkbox at the bottom of the World Limits rollout.

Programming the Ghost's Spawn Function

Open up your Project\Game\GameScripts\enemy.cs file in your default text editor. Edit the `enemyGhost::onLevelLoaded()` function to read like this:

```
function enemyGhost::onLevelLoaded(%this, %scenegraph)
{
  %this.startX = %this.getPositionX();
  %this.spawn();
}
```

Notice that we completely took out the `enemyMovement()` callout. You can delete the `enemyMovement()` function as well if you like, because we will not be using it anymore. This is because we are going to write a `spawn()` function that will handle enemy movements for us.

```
function enemyGhost::onWorldLimit(%this, %mode, %limit)
{
  if(%limit $= "left")
  {
  %this.spawn();
  }
}
function enemyGhost::spawn(%this)
{
  %this.setLinearVelocityX(getRandom(%this.minSpeed, %this.maxSpeed));
  %this.setPositionY(getRandom(%this.minY, %this.maxY));
  %this.setPositionX(%this.startX);
  %this.setCollisionActive(true, true);
  %this.setCollisionPhysics(false, false);
  %this.setCollisionCallback(true);
}
function enemyGhost::onCollision(%srcObj, %dstObj, %srcRef, %dstRef, %time,
%normal, %contactCount, %contacts)
{
  if(%dstObj.class $="BattyPlayer")
  {
  %dstObj.explode();
  }
}
```

The first function checks conditions to see if the ghost has reached the far left edge of its world limits, and if so it calls the `spawn()` function. The `spawn()` function sets the ghost back at a position absolutely set at the X axis where the ghost first started and at a random position on the Y axis somewhere between `minY` and `maxY`. It also sets up collision capabilities, and the final `onCollision()` function does a check to see if the ghost collides with the player character, and if so, it effectively kills the player.

However, `minY` and `maxY` are values that don't exist yet. We'll have to set them in the Level Builder, so save your enemy.cs file and return to the Level Builder momentarily.

1. Select the ghost image in the scene. Click and drag it to a position just outside camera view, where you want it to spawn.

2. Go to the Edit tab and scroll down to the Dynamic Fields rollout. Create two new fields, called `minY` and `maxY`.

The values that you should put in for these two fields are (a) the highest position on the screen where you want the enemies to appear, and (b) the lowest position on the screen where you want the enemies to appear. If you are confused how to get these values, click and drag your ghost image to the highest Y position, as you see in Figure 8.4. Then go to the Scene Object rollout (under the Edit tab) and see what the Y position reads and write it down. Then click and drag your ghost image to the lowest Y position and repeat. These are the values you should input. (Note: You can round the numbers to the closest whole number.)

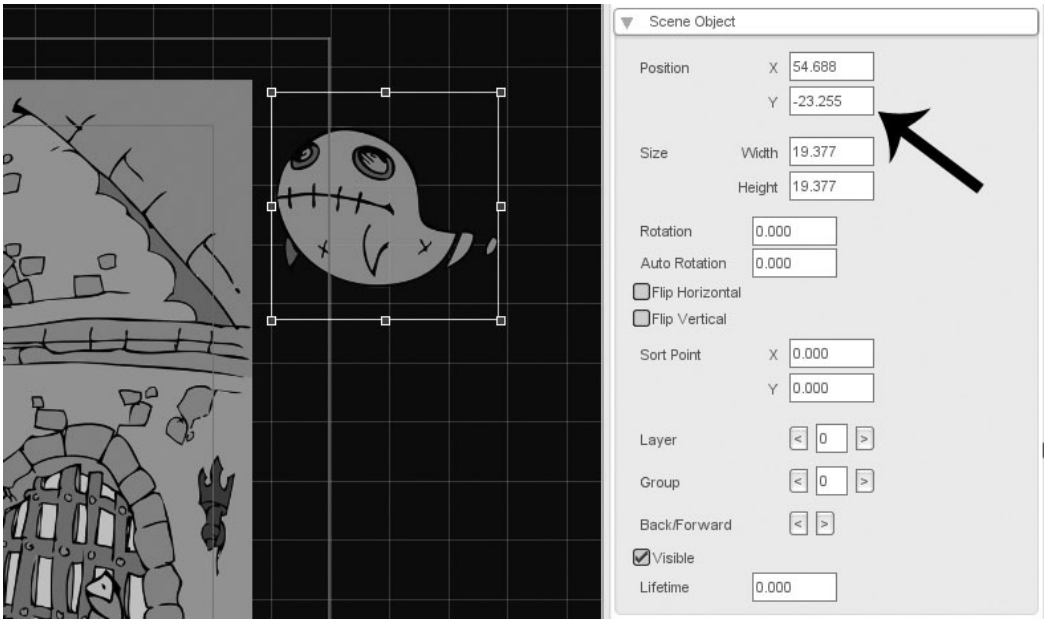


Figure 8.4
Click and drag your enemy all the way to the highest Y position and then look to see what its Y position is in the level.

Having the Ghost Destroy the Player

One thing we're missing is the player's `explode()` function. This is what will happen when the player character is destroyed from colliding with a ghost.

We will have `explode()` register that the player character is dead, make the player character inoperable for two seconds (which we have to schedule in milliseconds, so it will be 2,000 milliseconds), and then respawn. Respawning is simple. We reset the player character's position to its start position and enable it again.

Open the player.cs file and plug in these code snippets below your other scripts:

```
function BattyPlayer::explode(%this)
{
  %this.isDead = true;
  %this.setEnabled(false);
  %this.schedule(2000, "spawn");
}
function BattyPlayer::spawn(%this)
{
  %this.isDead = false;
  %this.setPosition(%this.startX, %this.startY);
  %this.setEnabled(true);
}
```

The variable `isDead` will serve as a way to tell if the player character is dead or not. In the above `explode()` function it will return a `true` value for `isDead`, and in the `spawn()` function it will return a `false` value for `isDead`.

But first we need to start the player out alive, not dead, so scroll back up until you find the `BattyPlayer::onLevelLoaded()` function and change it by adding the variable set to `false` like so:

```
function BattyPlayer::onLevelLoaded(%this, %scenegraph)
{
 $MeBatty = %this;
 moveMap.bindCmd(keyboard, "w", "BattyPlayerUp();", "BattyPlayerUpStop();");
 moveMap.bindCmd(keyboard, "s", "BattyPlayerDown();",
"BattyPlayerDownStop();");
 moveMap.bindCmd(keyboard, "a", "BattyPlayerLeft();",
"BattyPlayerLeftStop();");
 moveMap.bindCmd(keyboard, "d", "BattyPlayerRight();",
"BattyPlayerRightStop();");
 %this.isDead = false;
}
```

While you're here, let's add a couple of lines that will save the player's start position so that when Batty respawns, he'll respawn in the same place every time:

```
function BattyPlayer::onLevelLoaded(%this, %scenegraph)
{
 $MeBatty = %this;
 moveMap.bindCmd(keyboard, "w", "BattyPlayerUp();", "BattyPlayerUpStop();");
 moveMap.bindCmd(keyboard, "s", "BattyPlayerDown();",
"BattyPlayerDownStop();");
 moveMap.bindCmd(keyboard, "a", "BattyPlayerLeft();",
"BattyPlayerLeftStop();");
 moveMap.bindCmd(keyboard, "d", "BattyPlayerRight();",
"BattyPlayerRightStop();");
 %this.isDead = false;
 %this.startX = %this.getPositionX();
 %this.startX = %this.getPositionX();
}
```

Save your player.cs file when you're done.

Proofreading Your Program

So far your enemy.cs file should read like this:

```
function enemyGhost::onLevelLoaded(%this, %scenegraph)
{
  %this.startX = %this.getPositionX();
  %this.spawn();
}

function enemyGhost::enemyMovement(%this)
{
  %this.setLinearVelocityX(getRandom(%this.minSpeed, %this.maxSpeed));
}

function enemyGhost::onWorldLimit(%this, %mode, %limit)
{
  if(%limit $= "left")
  {
  %this.spawn();
  }
}
```

```
function enemyGhost::spawn(%this)
{
  %this.setLinearVelocityX(getRandom(%this.minSpeed, %this.maxSpeed));
  %this.setPositionY(getRandom(%this.minY, %this.maxY));
  %this.setPositionX(%this.startX);
  %this.setCollisionActive(true, true);
  %this.setCollisionPhysics(false, false);
  %this.setCollisionCallback(true);
}

function enemyGhost::onCollision(%srcObj, %dstObj, %srcRef, %dstRef, %time,
%normal, %contactCount, %contacts)
{
  if(%dstObj.class $="BattyPlayer")
  {
  %dstObj.explode();
  }
}
```

Your player.cs file should read like the following:

```
function BattyPlayer::onLevelLoaded(%this, %scenegraph)
{
 $MeBatty = %this;
 moveMap.bindCmd(keyboard, "w", "BattyPlayerUp();", "BattyPlayerUpStop();");
 moveMap.bindCmd(keyboard, "s", "BattyPlayerDown();",
"BattyPlayerDownStop();");
 moveMap.bindCmd(keyboard, "a", "BattyPlayerLeft();",
"BattyPlayerLeftStop();");
 moveMap.bindCmd(keyboard, "d", "BattyPlayerRight();",
"BattyPlayerRightStop();");
 %this.isDead = false;
 %this.startX = %this.getPositionX();
 %this.startX = %this.getPositionX();
}

function BattyPlayerUpStop()
{
  $BattyPlayer.setLinearVelocityY(0);
}

function BattyPlayerDownStop()
{
```

```
  $BattyPlayer.setLinearVelocityY(0);
}

function BattyPlayerLeftStop()
{
  $BattyPlayer.setLinearVelocityX(0);
}

function BattyPlayerRightStop()
{
  $BattyPlayer.setLinearVelocityX(0);
}

function BattyPlayerUp()
{
  $BattyPlayer.setLinearVelocityY(-15);
}

function BattyPlayerDown()
{
  $BattyPlayer.setLinearVelocityY(15);
}

function BattyPlayerLeft()
{
  $BattyPlayer.setLinearVelocityX(-25);
}

function BattyPlayerRight()
{
  $BattyPlayer.setLinearVelocityX(25);
}

function BattyPlayer::explode(%this)
{
  %this.isDead = true;
  %this.setEnabled(false);
  %this.schedule(2000, "spawn");
}

function BattyPlayer::spawn(%this)
{
```

```
  %this.isDead = false;
  %this.setPosition(%this.startX, %this.startY);
  %this.setEnabled(true);
}
```

Editing the Ghost's Collision Polygon

We've thus far set up a collision response that will occur whenever the ghost image crosses our Batty image, but we still need to set up a collision polygon around our ghost image so it will detect collisions accurately. You'll need to use the collision detection system any time you want to check if one of your game objects hits another. This can be used for all kinds of things, from making balls bounce off walls like in *Pong*, to lowering levers when the player reaches them.

Note

Batty already has a saved custom collision polygon around him, which you can see if you select him and click the Edit Collision Polygon button. All you need to do for this exercise is create a custom collision polygon for the ghost.

1. In your Level Builder, select your ghost. Once selected, hover over it and you should see five icon buttons appear above the object.
2. Click the first button in the set, the Edit Collision Polygon button (see Figure 8.5), to enter the Collision Polygon editing mode.
3. Your view should zoom in until the object you had selected is the only thing you see. Once in this mode, it is time to define the object's custom collision polygon.
4. Wherever you click your mouse, a new anchor point for your collision polygon is created. Go all the way around your object, keeping the boundary tight but not concave anywhere (as seen in Figure 8.6).
5. To exit out of Collision Polygon editing mode, click Save.
6. With your ghost still selected, go to the Edit tab and find the Collision rollout.
7. Here you will need to uncheck Send and Receive Physics and check Send Collision, Receive Collision, and Callback (as shown in Figure 8.7). Done! Save your level.

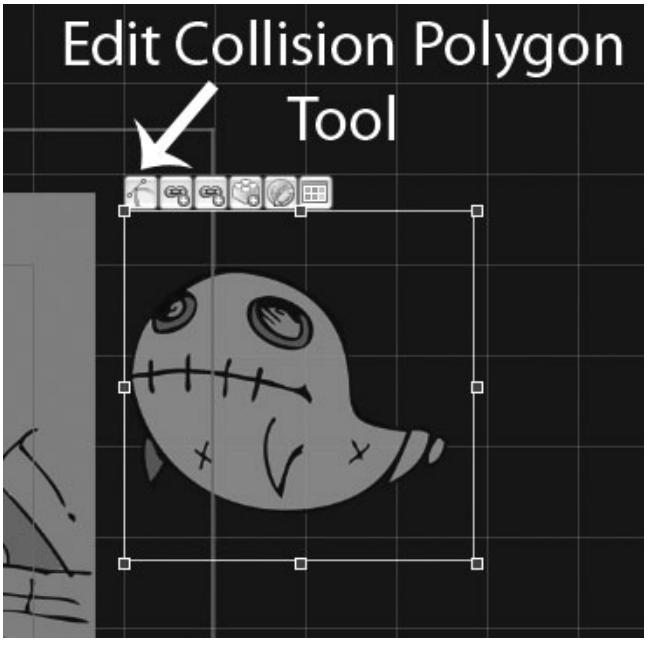


Figure 8.5
The Edit Collision Polygon icon button is the first one on the left.

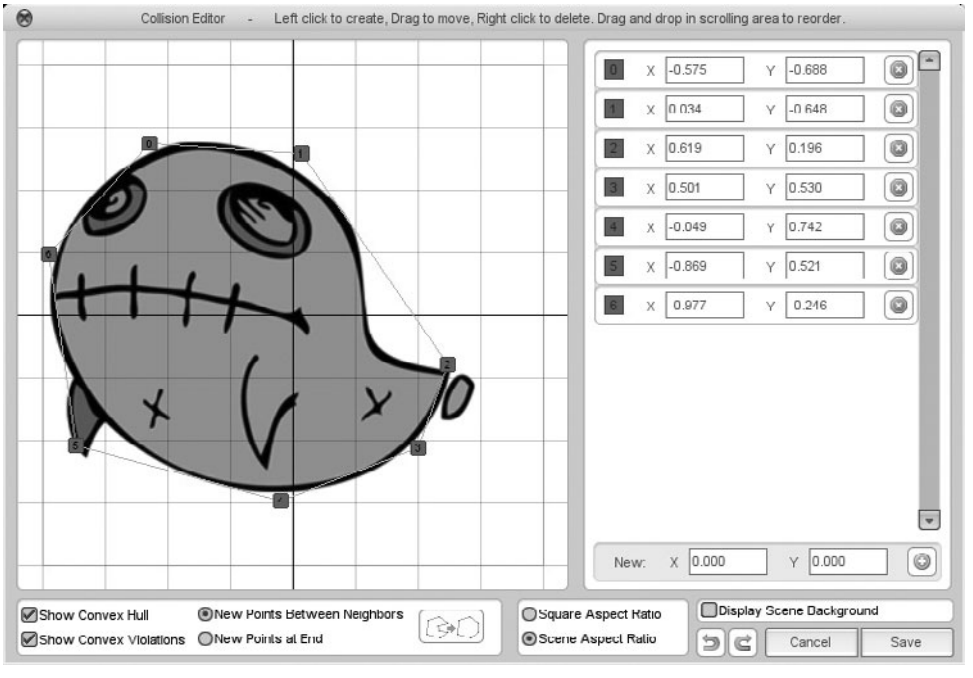


Figure 8.6
Trace a collision polygon around your ghost image by clicking to add anchor points.

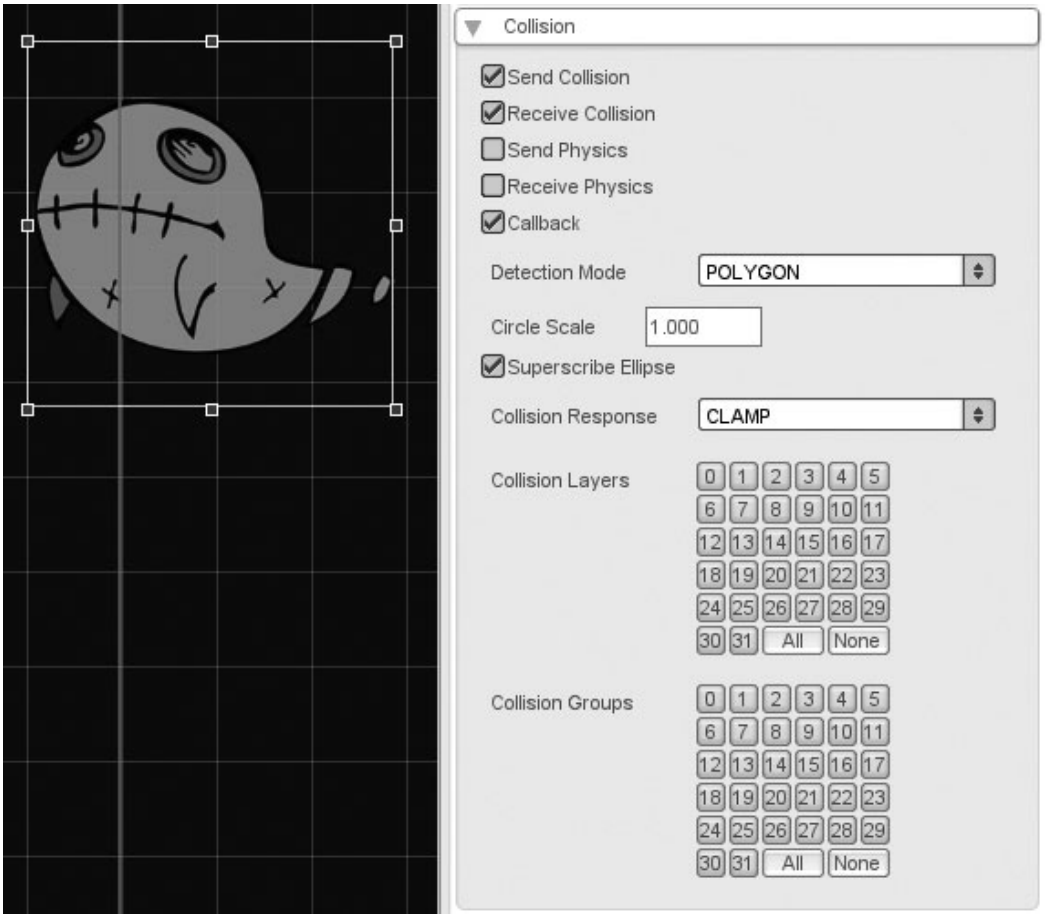


Figure 8.7
Check and uncheck these boxes in the Collision rollout.

Testing Your Level

Save your level and play it to see what happens. Your ghost should appear offscreen from the right, move to the left, disappear for a second, and then reappear from the right at a random location each time! The added randomness makes it look like each ghost is a different one from the next.

To make it look like Batty is fighting off hordes of opponents, select the ghost image in the Level Builder and press Ctrl + C (Win) or CMD + C (Mac) to copy the ghost instance to the Clipboard. Then press Ctrl + V (Win) or CMD + V (Mac) to paste the ghost instance from the Clipboard. Click and drag the new ghost instance to another location for its starting X position, if you like. You can

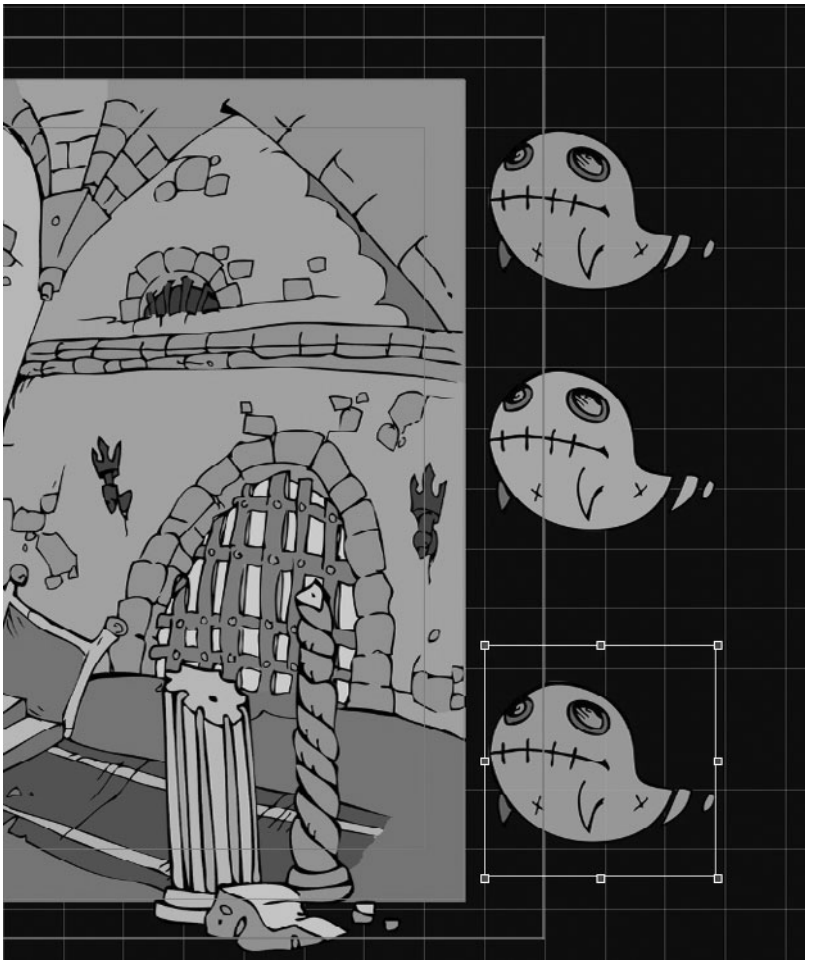


Figure 8.8
You can use the copy and paste method to add multiple ghosts.

repeat this step over and over, until you have a horde of ghastly ghosts, as shown in Figure 8.8.

Save and test your level again and see what happens. Gruesome Batty destruction does, that's what!

Having the Player Fight Back

Unfortunately, the player is defenseless against these ghosts. If he comes in contact with a single one of them, he disappears only to reappear back where he started two seconds later. Batty doesn't fight back! Now let's look at giving Batty a weapon.

Giving Batty a Ray Gun

We could make contact with Batty harmful so that if the ghosts ran into him they'd be destroyed—but that would be stupid, since their touch is the one toxic to him! We could have him spit globs of acid, which would be really nasty. Instead, we are going to give him a ray gun that shoots death rays.

1. Under the Create tab click the Create a New Image Map button.
2. Locate the raygun.png file. Click OK and the ray gun will become a sprite in the Static Sprites rollout under the Create tab.
3. From the Static Sprites rollout, drag and drop your ray gun image into your level on the left.
4. Once you have the ray gun placed in the level, resize it to fit with Batty, as seen in Figure 8.9, and move it into position.
5. With the ray gun image still selected, hover over it with your cursor until you see four icon buttons appear above it.

Figure 8.9
Set up the ray gun image in the castle environment.

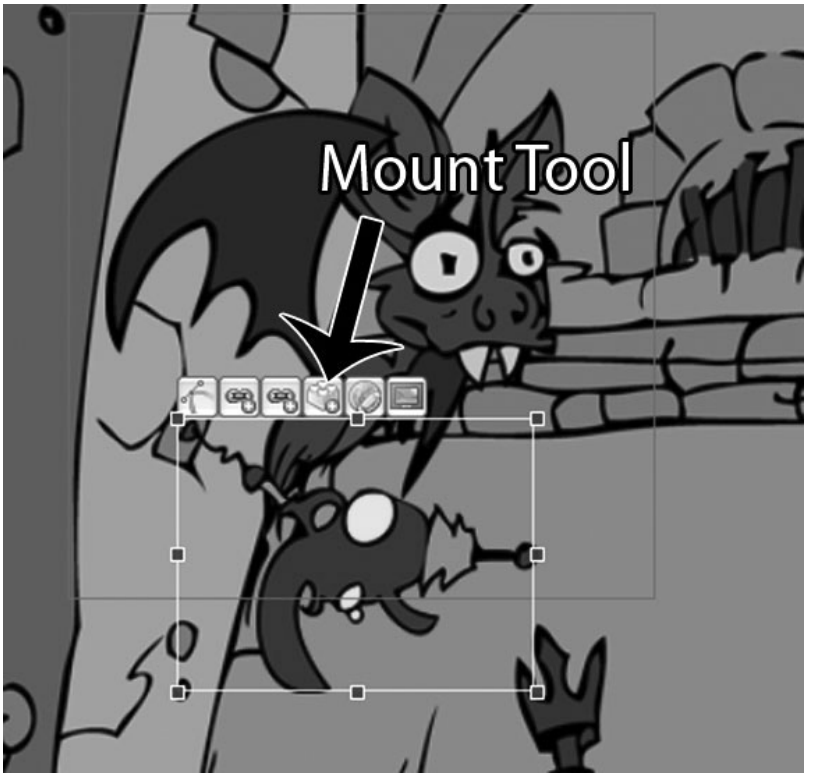


Figure 8.10
The Mount Tool icon button is the third from the left.

6. Choose the third button icon from the left (shown in Figure 8.10). This is the Mount tool. Once you click it, the object you had selected will be copied to your cursor.

7. Click on Batty's feet to complete the operation and mount the ray gun to the feet. It's that easy!

Creating Death Rays

First, before programming the death rays, we need to load the death ray source image into our level.

1. In the Level Builder, go to the Create tab and click the Create a New Image Map button.

2. Locate the deathray.png file. Click OK and `deathrayImageMap` will become a sprite in the Static Sprites section of your editor, under the Create tab.

3. Now select the `Batty` object and go to the Edit tab. Scroll down to find the Dynamic Fields rollout and add a new variable field, called `missileSpeed`. Set the `missileSpeed`'s value at or around 850.

Programming the Death Rays

Now open the player.cs file in your default text editor. Scroll down to where we did key map binding for the `BattyPlayer::onLevelLoaded()` function. You need to add a new key control:

```
moveMap.bindCmd(keyboard, "ctrl", "$MeBatty.createDeathray();","");
```

This binds the Ctrl key (for Win users, for Mac users it's CMD) to the function `createDeathray()`—which we now need to program.

Scroll down to the bottom of your player.cs file and add this new code:

```
function BattyPlayer::createDeathray()
{
  if(!%this.isDead)
  {
   %this.playerDeathray = new t2dStaticSprite()
   {
     scenegraph = %this.scenegraph;
     class = playerDeathray;
     missileSpeed = %this.missileSpeed;
     player = %this;
   };
   %this.playerDeathray.fire();
  }
}
```

This creates our death ray, as long as the player isn't currently dead, and sets the death ray's class and speed. Also, it creates fields that store the scenegraph and the player. After creating the death ray, it calls the `fire()` function for the `playerDeathray` class that we haven't created yet.

Now add the `fire()` function:

```
function playerDeathray::fire(%this)
{
```

```
  %this.setWorldLimit(kill, "1 2 3 4");
  %this.setLinearVelocityX(%this.missileSpeed);
  %this.setPosition(%this.player.getPosition());
  %this.setImageMap(deathrayImageMap);
  %this.setSize(83, 38); // approximately half size of original graphic
  %this.setCollisionActive(true, true);
  %this.setCollisionPhysics(false, false);
  %this.setCollisionCallback(true);
}
```

Wait! Where you see the `setWorldLimit()` callout, the numbers 1, 2, 3, and 4 are only placeholders! You have to put real values in them, but they will be different for everyone, depending on the size of your camera view. In order, they represent:

- **1**: The lowest X coordinate.
- **2**: The lowest Y coordinate.
- **3**: The highest X coordinate.
- **4**: The highest Y coordinate.

You can find these numbers fairly easily by looking under the World Limits rollout of your `Batty` object, because they should be the same.

The next function we will write deals with what will happen when the death ray collides with something:

```
function playerDeathray::onCollision(%srcObj, %dstObj, %srcRef, %dstRef,
%time, %normal, %contactCount, %contacts)
{
  if(%dstObj.class $="enemyGhost")
  {
  %srcObj.explode();
  %dstObj.explode();
  }
}
```

Basically, all we did was have the death ray check to see if it had come in contact with the enemy, and if so, both it and the enemy are immediately destroyed and removed from the level. Last thing we need to do is handle the destruction functions of both the death ray and the enemy object:

```
function playerDeathray::explode(%this)
{
```

```
  %this.safeDelete();
}
```

Save your player.cs file and switch to your enemy.cs file shortly. Add the following function to the bottom of your program code:

```
function enemyGhost::explode(%this)
{
  %this.spawn();
}
```

Rather than delete our ghost completely from the level (because then we'd have a rapidly dwindling population of bad guys to face), we simply tell it to respawn whenever it's been destroyed. Pretty sweet, eh?

Proofreading Your Program

Now your player.cs file should read like the following:

```
function BattyPlayer::onLevelLoaded(%this, %scenegraph)
{
 $MeBatty = %this;
 moveMap.bindCmd(keyboard, "w", "BattyPlayerUp();", "BattyPlayerUpStop();");
 moveMap.bindCmd(keyboard, "s", "BattyPlayerDown();",
"BattyPlayerDownStop();");
 moveMap.bindCmd(keyboard, "a", "BattyPlayerLeft();",
"BattyPlayerLeftStop();");
 moveMap.bindCmd(keyboard, "d", "BattyPlayerRight();",
"BattyPlayerRightStop();");
 moveMap.bindCmd(keyboard, "ctrl", "$MeBatty.createDeathray();","");
 %this.isDead = false;
 %this.startX = %this.getPositionX();
 %this.startX = %this.getPositionX();
}

function BattyPlayerUpStop()
{
  $BattyPlayer.setLinearVelocityY(0);
}

function BattyPlayerDownStop()
{
  $BattyPlayer.setLinearVelocityY(0);
}
```

```
function BattyPlayerLeftStop()
{
  $BattyPlayer.setLinearVelocityX(0);
}

function BattyPlayerRightStop()
{
  $BattyPlayer.setLinearVelocityX(0);
}

function BattyPlayerUp()
{
  $BattyPlayer.setLinearVelocityY(-15);
}

function BattyPlayerDown()
{
  $BattyPlayer.setLinearVelocityY(15);
}

function BattyPlayerLeft()
{
  $BattyPlayer.setLinearVelocityX(-25);
}

function BattyPlayerRight()
{
  $BattyPlayer.setLinearVelocityX(25);
}

function BattyPlayer::explode(%this)
{
  %this.isDead = true;
  %this.setEnabled(false);
  %this.schedule(2000, "spawn");
}

function BattyPlayer::spawn(%this)
{
  %this.isDead = false;
  %this.setPosition(%this.startX, %this.startY);
  %this.setEnabled(true);
}
```

```
function BattyPlayer::createDeathray()
{
  if(!%this.isDead)
  {
   %this.playerDeathray = new t2dStaticSprite()
   {
     scenegraph = %this.scenegraph;
     class = playerDeathray;
     missileSpeed = %this.missileSpeed;
     player = %this;
   };
   %this.playerDeathray.fire();
  }
}

function playerDeathray::fire(%this)
{
  %this.setWorldLimit(kill, "1 2 3 4");
  %this.setLinearVelocityX(%this.missileSpeed);
  %this.setPosition(%this.player.getPosition());
  %this.setImageMap(deathrayImageMap);
  %this.setSize(83, 38); // approximately half size of original graphic
  %this.setCollisionActive(true, true);
  %this.setCollisionPhysics(false, false);
  %this.setCollisionCallback(true);
}

function playerDeathray::onCollision(%srcObj, %dstObj, %srcRef, %dstRef,
%time, %normal, %contactCount, %contacts)
{
  if(%dstObj.class $="enemyGhost")
  {
  %srcObj.explode();
  %dstObj.explode();
  }
}

function playerDeathray::explode(%this)
{
  %this.safeDelete();
}
```

Your enemy.cs file didn't change much, but here's what it should look like:

```
function enemyGhost::onLevelLoaded(%this, %scenegraph)
{
  %this.startX = %this.getPositionX();
  %this.spawn();
}

function enemyGhost::enemyMovement(%this)
{
  %this.setLinearVelocityX(getRandom(%this.minSpeed, %this.maxSpeed));
}

function enemyGhost::onWorldLimit(%this, %mode, %limit)
{
  if(%limit $= "left")
  {
  %this.spawn();
  }
}

function enemyGhost::spawn(%this)
{
  %this.setLinearVelocityX(getRandom(%this.minSpeed, %this.maxSpeed));
  %this.setPositionY(getRandom(%this.minY, %this.maxY));
  %this.setPositionX(%this.startX);
  %this.setCollisionActive(true, true);
  %this.setCollisionPhysics(false, false);
  %this.setCollisionCallback(true);
}

function enemyGhost::onCollision(%srcObj, %dstObj, %srcRef, %dstRef, %time,
%normal, %contactCount, %contacts)
{
  if(%dstObj.class $="BattyPlayer")
  {
  %dstObj.explode();
  }
}

function enemyGhost::explode(%this)
{
  %this.spawn();
}
```

If you notice a terrible glitch during testing, or one of your routines is not working correctly, you should always come back to proofreading your code. It's easy to make a simple spelling slip-up, leave out a semicolon, or forget to close a bracket—and if you do so, it could cause cascading errors! So double-check your program code thoroughly.

Testing Your Level

Save your level and play it to see what happens. You should be able to fly around, up and down, and shoot death rays from your ray gun, as in Figure 8.11. When a death ray comes into contact with a ghost, it should immediately disappear. Meanwhile, you can dodge and weave around the oncoming ghost onslaught. Also, try out the speed boost, mapped to the spacebar.

The only things that might need tweaking later on down the line, if you want to experiment further, are the facts that the death ray only faces one direction, only shoots laterally in one direction, and that the destruction done to the ghosts is rather anticlimactic.

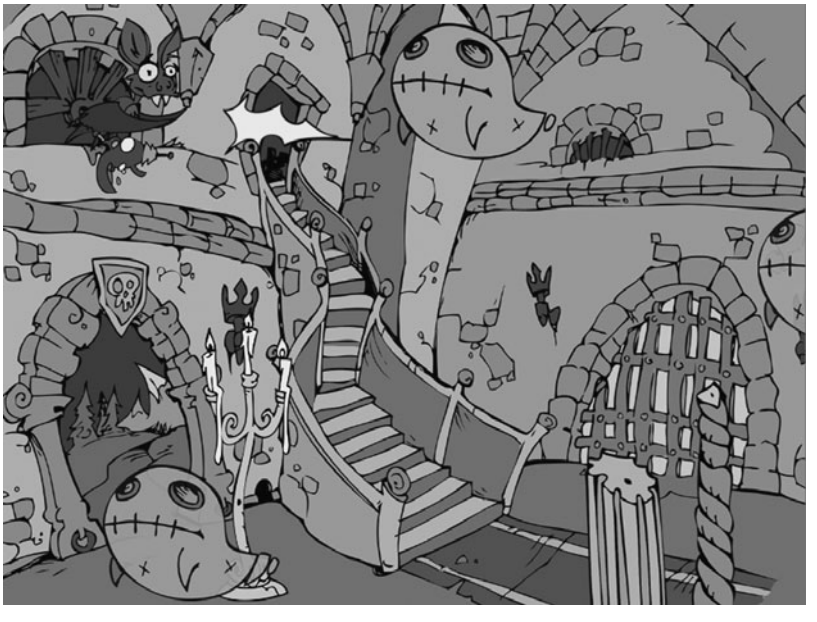


Figure 8.11
Shoot the ghosts with the ray gun.

Nuking the Ghosts

Let's make the destruction of the enemies more exciting. Right now they just sort of blink out of existence when hit with a death ray, which is fine for gameplay but not as satisfying as fragging ghosts should be.

Creating a Particle Effect

A *particle effect* is a group of similar graphics that combine to make an interesting visual effect. Particle effects are generated automatically according to a similar set of parameters that can be edited in Torque. Particle effects are commonly used for things like explosions, rain, smoke, and several other cool animated special effects. The Torque Game Builder particle system is incredibly flexible and includes a massive amount of features.

1. Go to the Create tab and click the Create a New Image Map button. Find the particle.png file and click OK. This will load our particle into the system. Now look under the Particle Effects rollout to see the particle effect `newEffect`.

2. Drag `newEffect.eff` onto the scene on the right. It starts emitting red lines (see Figure 8.12).

3. With the `newEffect` object selected, click on the Edit tab or double-click the `newEffect` object to go there.

4. Type **Fireball** into the Create Emitter input field and click the plus sign next to it. A new Emitter subsection appears in the Particle Effect rollout, called Emitter–Fireball.

5. Expand the Emitter–Untitled Emitter subsection and click the big Delete this Emitter button.

6. In the Emitter–Fireball subsection, open the Image drop-down list. Choose the `particleImageMap`, which is the particle image we just loaded into the system. This will be our emitter. An emitter is a component of a particle effect which will generally create one type of particle. Now you see some tiny fireballs shooting out in a circle. However, this looks silly compared to our ideal explosion.

7. Click the Save Effect button and name the new effect myExplosion.eff.

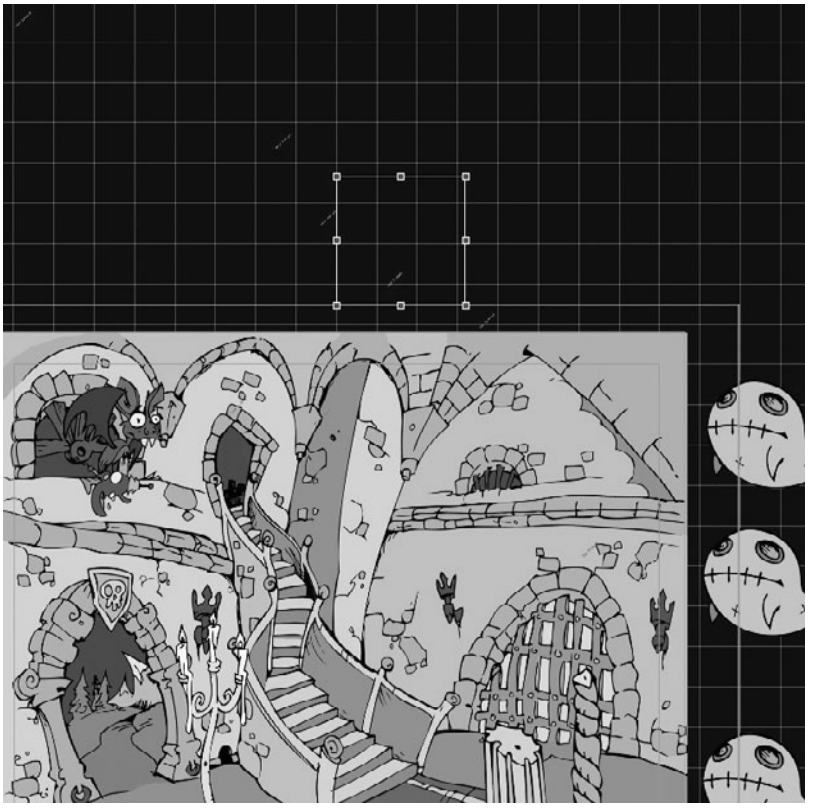


Figure 8.12
The `newEffect.eff` spews Test Texture graphics by default.

8. Click on the Edit Emitter Graph icon next to the Emitter–Fireball label and the graph editor dialog box will pop up. This editor is where we can alter our particles' properties over time.

9. Click the Current Field drop-down menu to see a list of options for changing particle properties. Select Size X Life from the list. This defines the size of our particles over their lifetime. The point on the far left is the start point.

10. Click on the start point and you'll see its starting value is set to 0, 1. The first numeral is the time and the second numeral is the value. The default says that at time 0, or when the particle is first created, its Size X Life value is 1. Drag that point up and down and you will see the particles change size. Leave the point at or around 0, 8.35.

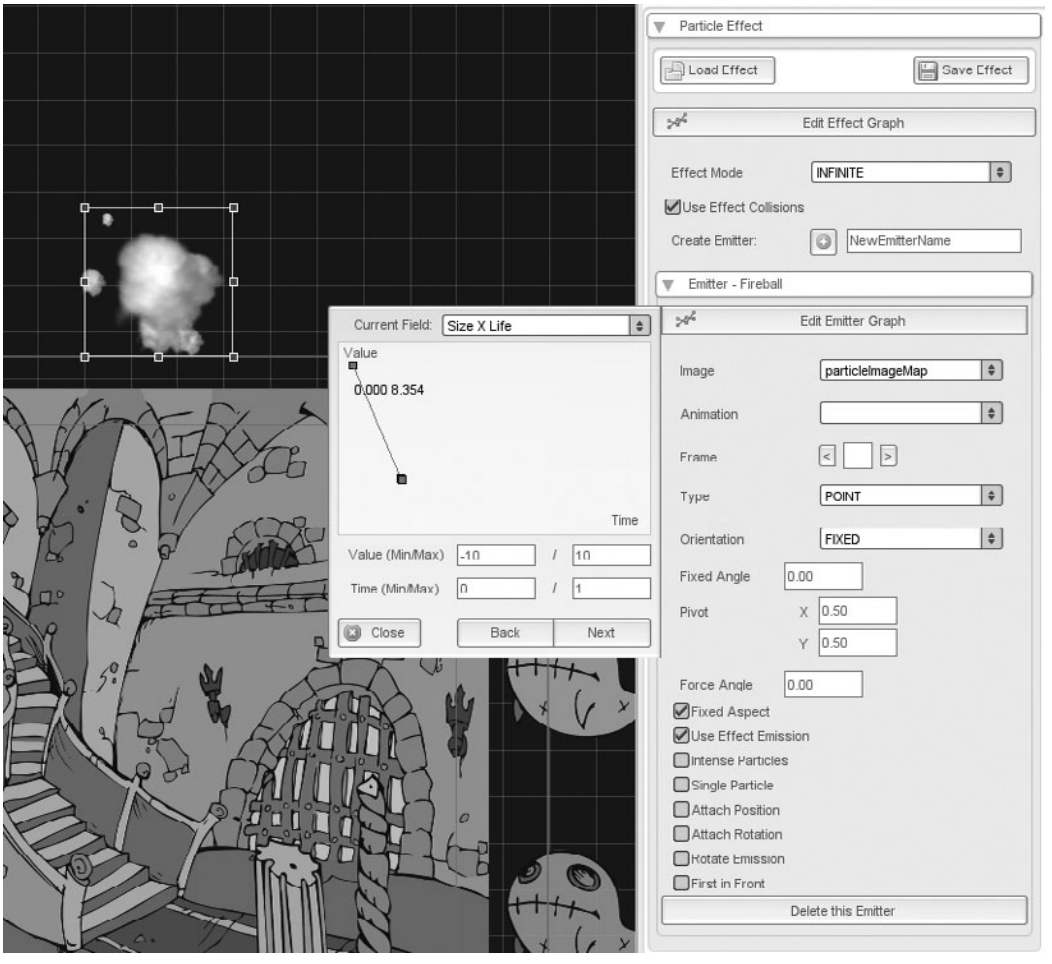


Figure 8.13
Setting points along the graph for Size X Life.

11. Click in the middle of your graph to make a new point and then drag the point until the fireball particles shrink down to nothing just outside the selection box. You have just created a simple graph of the particles' size, starting full size at time zero and then shrinking as time progresses. Compare your work to Figure 8.13. Note: if you make a point on your graph and don't want to keep it, you can delete it by right-clicking on it; however, you are not allowed to delete the starting point.

12. Click the Current Field drop-down menu to see a list of options for changing particle properties. Select Visibility Life from the list. This controls the particles' transparency.

13. Place three points along the graph editor and drag the points up and down until you reach your objective, which is to have the particle appear invisible at the start, quickly fade to full visibility in just a tic, then gradually fade back to invisible by the end point.

14. Instead of a continuous spray of particles, we want a massive explosion. To do so, go up to the main Particle Effect section and click on Edit Effect Graph to open the main graph editor, which changes properties for the whole kit-and-kaboodle.

15. Select Quantity Scale from the Current Field drop-down menu.

16. Place three points along the graph. The first point needs to be at or around 0, 20, the second needs to be at or around 0.25, 14.8, and the last point needs to be at or around 0.5, 5.5. This will make the effect produce really explosive particles and then stop. Compare your work to Figure 8.14.

17. Be sure to save your level.

Programming the Explosion

In the enemy.cs file, alter the `enemyGhost::explode()` function to look like this:

```
function enemyGhost::explode(%this)
{
  %explosion = new t2dParticleEffect()
  { scenegraph = %this.scenegraph; };
  %explosion.loadEffect("~/data/particles/myExplosion.eff");
  %explosion.setEffectLifeMode("KILL", 1);
  %explosion.setPosition(%this.getPosition());
  %explosion.playEffect();
  %this.spawn();
}
```

Save your enemy.cs file and exit. When you have reloaded your level in the Level Builder, test it out and see how your new particle effects work! Watch the ghosts disappear in a flaming ball, like in Figure 8.15. Now, isn't that more satisfying?

Adding Audio

Sound makes everything come alive. It stamps the heartbeat for our culture and provides us with an aural experience. It can support the story of a game or shape its soundtrack. Try playing a video game with the TV or computer speakers on mute, and you'll realize just how disappointing the experience quickly becomes.

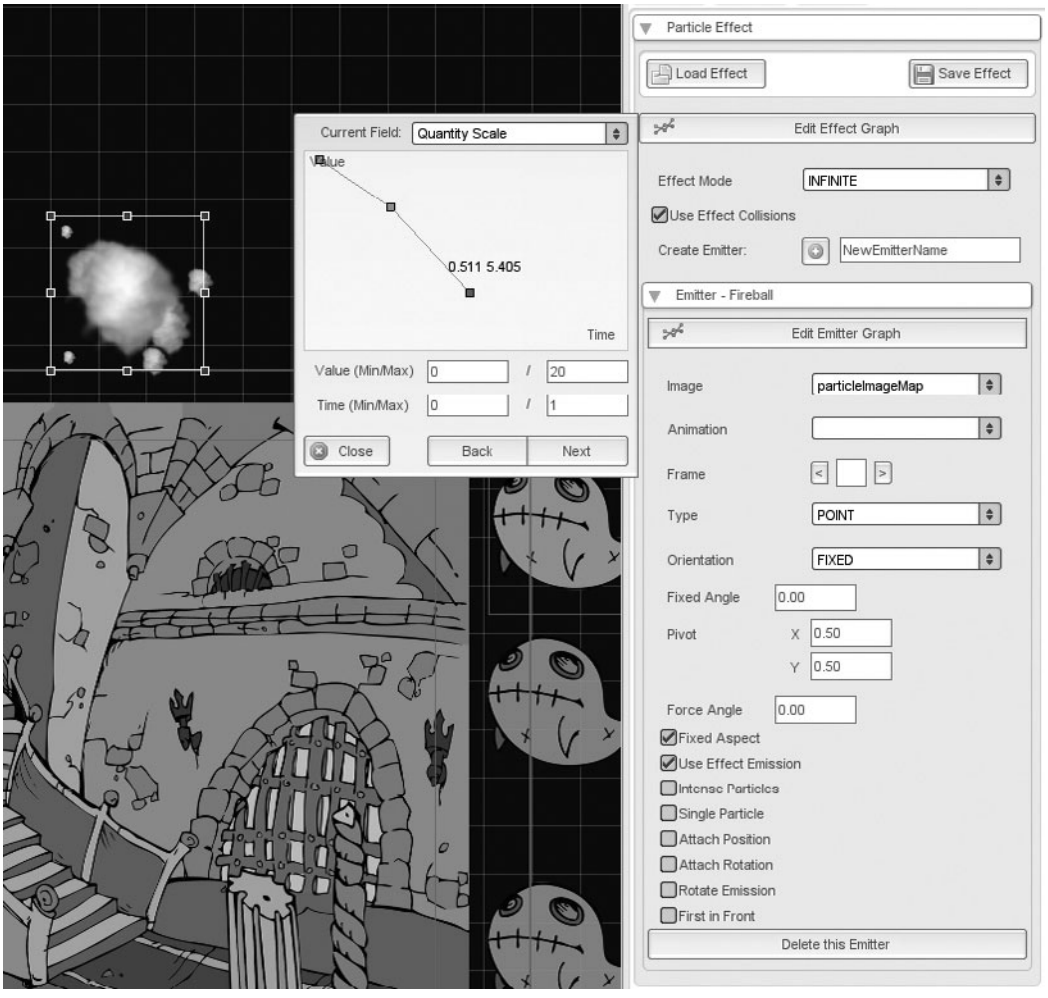


Figure 8.14
Setting points along the graph for Quantity Scale.

Hollywood Sound

Tip

"Similar to film, music in games establishes the emotional context—providing the player with an emotional connection to the visuals on the screen. Music feeds into the gameplay and, if implemented well, the music and visuals will fully immerse the player in the game environment . . . Music is an essential component in elevating the game experience."

—Greg O'Connor, founder of Music4Games.Net

Figure 8.15
Bye-bye, ghosts! Batty's got a gun.

The same rules apply to games as they do with blockbuster movies, when it comes to the audio. This is because electronic games follow the wake of TV and cinema, so game developers can learn a lot from the way filmmakers use sound. Sound can support a narrative, telling the story directly or indirectly enhancing the overall occurrence. In games, sound can also offer players the feedback they need for a truly satisfying experience.

A Quick History Lesson

The movie industry started seriously using sound in the 1920s, but the process was incredibly difficult back then. Because of the limitations of the sound cameras, actors often had to be experienced theatrical performers and capable of projecting their voices to be heard on the final recording. A good number of silent film actors lost their jobs about this time, because audiences discovered they had poor speaking voices or undesirable foreign accents.

Today's recording processes are a whole lot better, and most of the work goes straight to digital tracks, which can be edited for even more solid quality. Actors can act more naturally and underplay their roles if they decide to and still be heard on the final track. Unfortunately, many directors are becoming more and

more dependent on the sound engineers to make the actors' voices sound clear in post-production, even when the directors can't hear them on set. This can be sometimes frustrating for sound engineers. Though directors can frame a shot to effectively remove a sign or offensive place from being seen on camera, the production crew has little to no control over random background noises, such as airplanes taking off overhead or construction workers off-camera.

Sound delivery has also vastly improved over the last few years. It started out mono and only as recently as the 1970s went to stereo. Today's modern audiences are used to hearing their TVs, MP3 players, and radios in stereo. Not only that but many people rock out with Dolby surround sound and amplified bass and expect to hear top-quality sound. The increased resolution of HDTV images requires a similarly improved high-definition sound track.

The technology continues to advance, and even the game industry must stop and think about sound critically during development.

Foley Sound

When you hear a door open and close or a scream at the cinema, you may not realize it at the time, but the sound is not a real sound. It's not even an original recorded sound. Most sounds you hear in movies and TV shows are accurate representations of real sounds dubbed to enhance the aural experience.

Sound engineers construct the sounds in post-production, utilizing many pieces of sounds that they mix together in software programs to create seamless creations. Many of the sounds come from sound effects libraries. This is why so many screams and door shuts sound the same, from one movie to the next.

Other sounds are custom recordings done in studios. They are called *Foley sounds*. Foley artists record custom sound effects that emphasize sounds that should be heard in context. For instance, a Foley artist might pulverize a watermelon to sound like a head splat, ruffle a straw broom to emulate someone walking through bushes, or shake a sheet of corrugated tin to record rumbling thunder. These are similar-sounding noises that, when mixed in an audio program, can add a great effect to the video actions.

Sound Basics

The Greek mathematician Pythagoras not only delivered us a triangle but also discovered the octave and came up with numeric ratios connected to harmony. Galileo formed many of the scientific laws of sound. Heinrich Hertz gave us the

Hertz, and Alexander Graham Bell gave us the decibel. Since the 1600s there have been numerous efficient advances in the study of sound, resulting in several sound laws.

Sound Laws

Sound comes in the shape of vibration waves. Some sound waves can actually travel at frequencies so high or low that we humans can't hear them at all, but some animals can. Think of the dog whistle, which is normally so high that humans can't hear it.

What follows are the basics of sound laws:

- A *sound wave* moves pretty much in a straightforward line.
- *Pitch* refers to how fast the sound wave vibrates, also known as the sound *frequency*.
- Humans can detect sound frequencies by a 2:1 difference, so many of our music notes are on a scale of 2.
- The term *Hertz* (Hz) comes from a unit of frequency equaling one vibration per second.
- *Intensity* is the volume or how loud the sound comes across, also called the *amplitude*. Intensity can be measured by the *decibel* (dB).
- The increase in intensity of a specific sound wave is known as the *gain*.
- *Timbre* is the waveform or accuracy of the sound frequency. Timbre is different for every instrument and every voice, and it reflects a change in quality that is not dependent on intensity or frequency.

Influencing Factors on Sound

The outside factors that influence sound include space, time, and situational events. As with many of the other elements of media aesthetics, these factors can overlap and become random influences.

Space's Impact on Sound Location defines how a sound is carried, for several reasons.

When trying to match up video images that appear closer to the screen, the sound must sound closer, and when there are long shots, the sounds must come from farther away. Close sounds take the spotlight, often sharing more presence than the softer background noises. This involves the scientific principle of the *Doppler Effect*, which states that as a loud sound source comes closer to the listener, the higher the sound gains, and the farther away the sound source moves from the listener, the softer the sound gets until it fades away completely.

Stereo sound makes it possible to hear sound relative to onscreen positioning through the use of multiple sound channels. For instance, most of the player character's dialogue will come from the front forward-facing speaker in 5.1 Dolby surround sound, but if you show a comrade shouting for the player to catch up and show him slightly off-screen to the left, you'd best make sure that the sound comes from one of the right-side speakers and is softer than the dialogue.

Another way that sound is impacted by location is dampening effects. As an example, snow is a sound dampener because of how closely packed ice crystals form. Thus a scene set in a wintery wonderland would have an almost hushed feel, and any sounds heard there would have a muted appeal. Sound in an underwater setting would also have a muted appeal, because sound waves find it much more difficult traveling through water.

Time's Impact on Sound Time, also, has a few very important influences on sound. It can be reflected, predictive, or an important leading motif.

Time-reflected sounds are those that alter with different times of day and varying climates. One backyard scene can have a vastly dissimilar soundscape based upon what time of day it is and what the weather's like. If it is cloudy, storming, and near midnight, the sounds would be more hushed than they would if it is sunny and warm at midday and the birds are all chirping away.

Predictive sounds involve the placement of certain sounds before an event actually takes place. An example of a predictive sound is if you are building a fish-feeding game and you want a giant barracuda to come across the screen, gobbling up everything in its place. As the developer you might decide to warn the player through sound effects. So you might have a warning noise, like a sudden discordant bell, strike just seconds before the barracuda creeps across the screen.

The other special use of sound dealing with time is the *leitmotiv*, which is German for "leading motif." Some games play a short music piece every time a specific boss monster or enemy shows up, letting the player know that it's time to rumble.

The leitmotiv for an innocent damsel in distress, on the other hand, might be more cheerful and airy. Some role-playing game developers even stick a personal soundtrack to each and every one of the non-player characters (NPCs) so the players can tell them apart.

Situational Impacts on Sounds Sounds can also describe specific situations or be affected by outside events that help put the listener into the scene. This is often done by layering sounds to make the setting truly come alive.

Let's look at an example. Say that you wanted to build a swamp setting. You have it muted because of location. You have it at night, so the time-reflection comes into play. Lastly, you would think about all the external situations that would make up the soundscape of a swamp. Perhaps you need to add crickets chirruping softly, cat tails rustling faintly, and the random croaking of a bullfrog. Perhaps you could further add a steady dripping of dampness and the distant rolling of thunder. All of this—and any more details your imagination or personal experience can offer—will add to the scene and bring it to life for the listener.

Sound in Games

Games use sound in three very important ways: as sound effects, for music soundtracks, and for voiceovers.

Sound Effects

Sound effects, sometimes abbreviated as SFX, are often short recorded sounds that are interjected relative to visual effects to enhance the whole experience and give aural clues to what's going on onscreen. Event-based sound effects serve as feedback for the player to the actions they input.

Music

Game music is a means of telling the players how they should be reacting to the images on the screen, kind of a mood compass, in other words. The music shifts to lighter tones for romantic or comic scenes and drops to deeper thrums when danger looms. A game soundtrack consists of a score and individual songs. The score refers to the instrumental music composed for the game with the intention of creating mood and atmosphere, while the songs are featured periodically, often in cutscenes.

Voiceovers

Voiceovers are done by artists recorded reading dialogue and narration scripts in a recording studio for purposes of providing spoken dialogue and narration in games. Voiceover artists (or VO artists) come from all walks of life, some of them accepted actors looking to supplant their income with off-screen work, some of them regular voice actors who specialize in doing characterizations, and some independents looking to break into the industry. Regardless of where they come from, VO artists must capture the character they're speaking as, including emotion, accent, and inflection.

Tip

"In games voice acting, each emotion—such as laughing, crying, screaming—has to be recorded. Each 'action' also has to be recorded; we call these 'efforts'—and they include punching, getting punched, sighing, getting attacked, performing full combat, dragging bodies, and being dragged . . . The energy it takes to voice act in a game is *endless*. It's a surprisingly physical job. We voice actors throw ourselves into the moment . . . so when we're being chased, we're running in place—huffing and puffing. After a four-hour session of screaming and really getting into the actions and emotions of what the game requires, we feel pretty spent!"

—Hope Levy, voice of Rebecca Chambers in *Resident Evil*

Using Sound Libraries

The companion CD that comes with this book has some royalty-free sound effect samples from Mojo Audio (http://www.mojoaudio.com). There are explosion sounds, sci-fi weapon sounds, magical spell sounds, and more. You can use the samples as your personal sound effects library until you get your own recordings made, or purchase the full audio packs from Mojo Audio's website.

What follows are online sound effects libraries, some economical and some totally free. Most offer sounds in .wav or .mp3 file formats.

- **Acoustica** (http://www.acoustica.com/sounds.htm)
- **Freesound Project** (http://freesound.iua.upf.edu)
- **Koumis Productions** (http://www.koumis.com/soundfx.htm)
- **Ljudo** (http://www.ljudo.com/default.asp?lang=tEnglish&do=it)
- **Media Tracks Free Sound Effects** (http://www.media-tracks.com/default.asp?ID=5)

- **Sound Effects Library** (http://sound-effects-library.com)
- **Stonewashed SFX** (http://www.stonewashed.net/sfx.html)

You might also want to take a good look at the independent music scene and license music from unsigned artists. Many indie musicians are starting to ally themselves with game developers. Several online resources include:

- **Artist Launch** (http://www.artistlaunch.com)
- **CD Baby** (http://www.cdbaby.com)
- **Indiespace** (http://www.indiespace.com)
- **MP3.com** (http://www.mp3.com)
- **Music4Games.net** (http://www.music4games.net)
- **SoundClick** (http://www.soundclick.com)

However, if you feel up to it, you can record your own sound effects and music scores by starting your own recording studio.

Mixing Your Own Audio

If you plan on recording and editing your own Foley sounds for your game, then here are a few suggestions:

- Set up a home sound studio by selecting someplace that can be sound-proofed, cutting out most outside noise. Short of buying expensive commercial eggshell panels to cover your walls and ceiling, you can drape wool blankets over windows and doors or add bookshelves to absorb sound waves.
- Pick a free or cheap sound-editing software program to perform audio mixes. Audacity, NCH Swift Sound, and Slab (Linux only) are totally free sound-editing programs you can find for download online, and there are several cheap alternatives as well. Use Google or another search engine to find the best options when looking for sound editors.
- Find a really good microphone. For killer vocal recording, you should consider a condenser microphone, which uses an electronically charged

stretched diaphragm over a thin plate to capture fluctuations into electric currents. It's more fragile and less likely to handle abuse than other mics, but it does output the best quality recording.

- Practice, practice, practice! Don't accept your first recording as "golden."

Digital Sound

It is important before you get started mixing and using digital audio files that you understand what digital sound files are called, what compression of these files means, and some of the keywords to use when talking about digital sound mixing.

Sound File Formats Computer-based sound editing generally involves one of three digital audio file formats: WAV files, MP3 files, and OGG files. WAV files are uncompressed audio files, while MP3 and OGG files are compressed.

Uncompressed Audio *WAV files* are generally uncompressed files, meaning that they can be quite large in size and sound pretty good. The quality of a WAV file is determined by how well it was originally recorded or converted. Usually you will want to work with WAV files for sound effects, but they can take up quite a lot of room in memory, which is not efficient when publishing your project later on for distribution.

Compressed Audio *Compression* restricts the range of sound by attenuating signals exceeding a threshold. By attenuating louder signals, you limit the dynamic range of sound to existing signals. Imagine that the audio file is a piece of paper with sheet notes on it. Compressing means you literally wad up the piece of paper into a tiny ball. To listen to the compressed audio files, you have to use a device like an MP3 player to un-wad and smooth out the piece of paper. The most popular compressed audio file on the market right now, due mostly to the popular consumerism of iPods and other MP3 players, is the MP3.

MP3 stands for Moving Pictures Expert Group, Audio Layer 3, and is a variation of the MPEG file. It started in the 1980s by the German Fraunhofer Institut. In 1997, the first commercially purchasable MP3 player was created, called the AMP MP3 Playback Engine, which was later cloned into the more popular Winamp software by college students Justin Frankel and Dmitry Boldyrev. Napster and its gangbuster follow-up MP3 file-sharing services blew the lid off the MP3 boom, making it the number one compressed audio file format on the Internet.

Ogg Vorbis, or simply OGG, is the format of choice on Linux systems and AIFF is preferred for Macintosh.

OGG files use a different, and some say better, encoding process to compress audio. If you've never heard of OGG files before, check out http://www.vorbis.com for more information.

AIFF stands for Audio Interchange File Format, and it was co-created by Apple Computer in 1988 based on the Electronic Arts' Interchange File Format (IFF) and most commonly used on Apple Macintosh computers since. With the development of the Mac OS X operating system, Apple quietly created a new type of AIFF which is, in effect, an alternative format.

Using Audacity

A great open-source program, which won't cost you a red cent under the GNU General Public License, is Audacity, seen in Figure 8.16. The program is on the companion CD, or you can download it from the Internet at http://audacity.sourceforge.net.

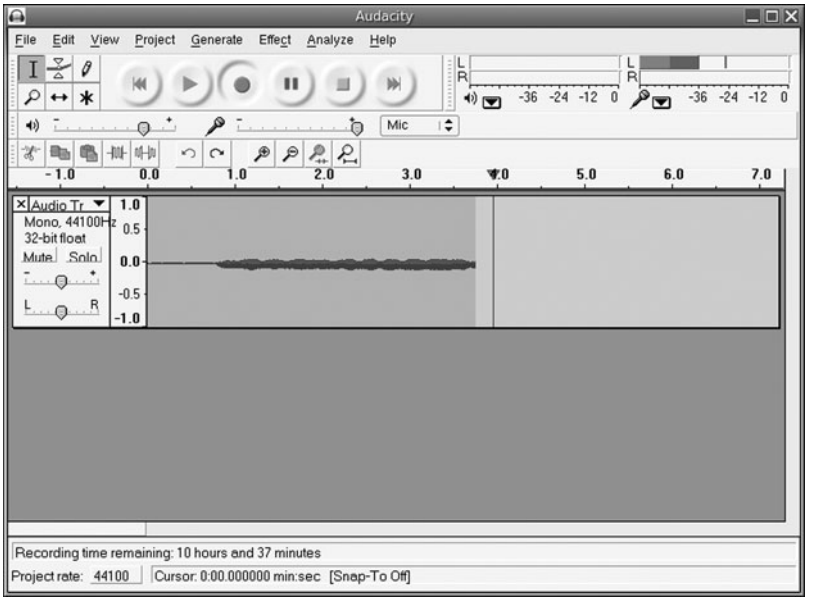


Figure 8.16
Audacity, from Sound Forge, is a user-friendly free audio editing program.

Record Sound After installing Audacity, open the program and let's record some sound. Click the Record button. The program will start recording from your microphone, so if you don't have a microphone yet this exercise will be pointless. As Audacity records, you can see the progress and waveforms as you speak into the mic.

When you're through recording, you can click the Stop button. Play back your recording by clicking the Play button. If you can't hear anything, but you see a waveform in Audacity, make sure you have the volume turned up on your computer and speakers. If everything looks like it's working correctly, but you still can't hear any sound, check the microphone level in the Mixer Control in Audacity, and you might raise it up a notch. Other problems that can arise include using a bad mic, plugging the mic into the wrong slot of your sound card, or the sound recording/playback volume defaulting to mute. Before panicking, check all these things.

Edit Sound Notice that if you didn't speak into the mic right away, you will have a period of "dead air" before the sound wave you made and another chunk of "dead air" after, as you were reaching to click the Stop button. This is alright but not optimal for sound recording purposes. You have to edit it.

Place your cursor over to one side of the portion of the waveform you want to remove and then click and drag across it to the other side, highlighting the area you want eliminated. Then select Edit > Delete. The selected portion will be removed from the waveform.

Notice that under the Edit menu you also have the option to Cut and Paste, which will come in handy when you mix multiple sound tracks together.

Play back the audio to make sure you didn't remove too much or not enough of the waveform. Eventually you will have finished your work.

Adding Effects Highlight the entire waveform and experiment with the effects that ship with Audacity. Click on the Effect menu to play around with any of the effects. You can play back the sound to hear how each effect changes the recording. Simply select Edit > Undo to remove the effect after hearing its influence.

Exporting Your Audio Lastly, you need to save the sound you've recorded and edited as a digital audio file format you can use later. Go to File > Export as WAV and name your file. Save it somewhere convenient for now, such as your desktop, then browse to where you placed your new audio file and open

it in your operating system's default media player. Listen to your newly made sound as it exists now.

Note that you can export your audio file as an MP3. Unfortunately, this requires a special plug-in not located on the CD with Audacity. The Audacity MP3 encoder can be found through a brief web search, if you so desire to export MP3 files.

Sound in the Torque Game Builder

The Torque Game Builder uses OpenAL for sound support, plus you can look online for further resources to use alternative libraries. Sound is one area that TGB makes look difficult at first glance, but it turns out to be easier in execution. The last thing I'll show you before we close is how to add some music and special fire noise when Batty shoots the ray gun.

Coding Audio Descriptions

Create a new text file in Game\GameScripts folder and call it audioFX.cs. In audioFX.cs type in the following:

```
new AudioDescription(AudioNonLooping)
{
  volume = 1.0;
  isLooping = false;
  is3D = false;
  type = $GuiAudioType;
};
new AudioDescription(AudioLooping)
{
  volume = 1.0;
  isLooping = true;
  is3D = false;
  type = $GuiAudioType;
};
```

This code simply sets up two new audio descriptions, one for looping sounds and one for non-looping sounds, or sounds that will only play once through.

The `AudioDescription` datablock tells the computer whether the sound is meant to be 2D or 3D (3D sound has a dynamic environmental source and a Doppler effect, which means the closer you get to its source, the louder it gets), and it also tells the computer if the sound is meant to loop, and if so how many times it

loops. Lastly, the AudioDescription datablock always sets the sound to a specific sound channel. TGB comes with specific sound channels, which are often dedicated to certain tasks:

- **Channel 0**: $DefaultAudioType
- **Channel 2**: $GuiAudioType
- **Channel 3**: $SimAudioType

Coding Audio Profiles

Next, we have to add two audio profiles, one for each of the sounds we are adding in this exercise. Add this bit of code just below your two audio descriptions:

```
new AudioProfile(bgAudio)
{
  filename = "~/data/audio/song.wav";
  description = "AudioLooping";
  preload = true;
};
new AudioProfile(fireAudio)
{
  filename = "~/data/audio/shoot.wav";
  description = "AudioNonLooping";
  preload = true;
};
```

The AudioProfile datablock defines the sound that will be played. The AudioProfile tells the computer what sound file will actually be used for the sound. It also answers the question of whether the sound file should be preloaded. Preloading sounds helps when certain sounds would take too long to load in real time. The AudioProfile links itself to one AudioDescription.

Save audioFX.cs and close it. To execute it you will have to add exec("./audioFX.cs"); to your startGame function in the main game.cs file, right beneath the exec("./enemy.cs"); line.

Coding Dynamic Sound

We've defined an AudioDescription and AudioProfile for two files now, but they don't play by themselves. We have to find the most congenial area in our program to call them. If you place audio directly into your game levels, they are

called *static sound*, because they sit as a permanently placed sound source within the game level. On the opposite end of the spectrum, a sound that is called from the program code and placed during the run-level process is called *dynamic sound*. We'll add our sounds as dynamic sounds.

First, let's play our background music. The best place to launch the music is within the `BattyPlayer::onLevelLoaded()` function, so open your player.cs file and modify that function as follows:

```
function BattyPlayer::onLevelLoaded(%this, %scenegraph)
{
 $MeBatty = %this;
 moveMap.bindCmd(keyboard, "w", "BattyPlayerUp();", "BattyPlayerUpStop();");
 moveMap.bindCmd(keyboard, "s", "BattyPlayerDown();",
"BattyPlayerDownStop();");
 moveMap.bindCmd(keyboard, "a", "BattyPlayerLeft();",
"BattyPlayerLeftStop();");
 moveMap.bindCmd(keyboard, "d", "BattyPlayerRight();",
"BattyPlayerRightStop();");
 %this.isDead = false;
 alxPlay(bgAudio);
}
```

Here's the fire sound we'll add to the `playerDeathray::fire()` function:

```
function playerDeathray::fire(%this)
{
  alxPlay(fireAudio);
  %this.setWorldLimit(kill, "1 2 3 4");
  %this.setLinearVelocityX(%this.missileSpeed);
  %this.setPosition(%this.player.getPosition());
  %this.setImageMap(playerDeathrayImageMap);
  %this.setSize(12, 2);
  %this.setCollisionActive(true, true);
  %this.setCollisionPhysics(false, false);
  %this.setCollisionCallback(true);
}
```

Note

Remember that where it says `"1 2 3 4"` after `%this.setWorldLimit`, your numbers will be different based on the size of your project's camera view!

Save your player.cs file and return to the Level Builder. Test it out, and see what happens. You should be able to hear your song.wav file playing over and over in the background, probably making the Crypt Keeper roll over in his grave, and when you shoot your ray gun, you should hear the shoot.wav file play once.

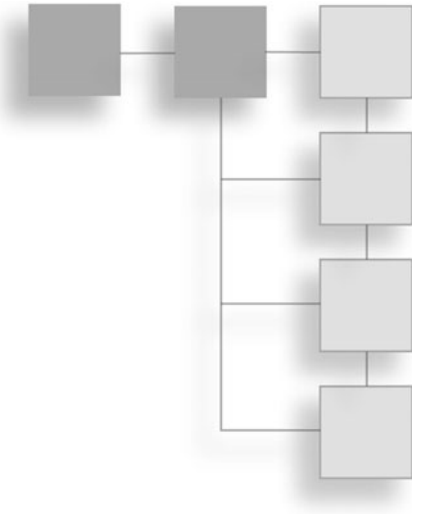
Review

After reading this chapter, you should understand the following:

- How to add an enemy that moves across the screen
- How to cause enemies to spawn at random positions
- How to make enemies destroy the player if they touch him
- How to give the player a weapon to shoot enemies with
- How to add background music and weapon-fire sound effects

This page intentionally left blank

CHAPTER 9



Getting Your Game Out There

In this chapter, you will learn:

- To advertise yourself by perfecting your presence and developing a gimmick
- To pitch a game proposal to possible publishers
- To advertise your games on the web by creating and maintaining a website
- How to use online community accounts to get your games noticed
- What to think about when it comes to your future goals and education

Your games can be about anything you want. The options are truly open-ended with the only confines being the scope of your imagination and your practiced proficiency with the Torque Game Builder. Make sports games, checkers, and fantasy role-playing games . . . Whatever drives you, you can build, given patience and hard work.

Besides building games to your heart's content, here are some examples of things you can do with the skills you have learned:

- You could make games to distribute online for people all around the world to download and play at their whim.

- You could make a private portfolio of games you have made to help you get into a good game design school.
- You could display games you've designed to publishers or developers to get your foot in the door in the industry.
- You could even rope some of your friends into joining your game design team and make games together that you can play.

Whatever you want to do to get the world to notice you and your game, we'll look at the how in this chapter.

Where to Go Now

Eventually you'll reach the end of creating your first homebrewed game with the Torque Game Builder and want to package and advertise it. When you get to that point, you may have a few friends who want to play it, but to really get the most people playing your game you're going to have to be clever . . . oh, so very clever! But first, you'll need to know how to package the game and get it ready for people to play.

The Packaging Utility

The Torque Game Builder Packaging Utility provides a highly efficient and extremely easy way to fabricate TGB distributable game packages. This means that when you are ready to package a consumer and non-TGB-licensed-owner build of your game, you can simply fire up the Torque Game Builder Packaging Utility, click a few buttons and options, and click the Build button to complete the process. Then, everything you want is copied into a convenient folder that you specify, leaving you with your source game directory untouched. The reason the TGB Packaging Utility leaves your source game directory untouched is so that you can continue tweaking it as you like, even after creating a published copy of your game. This makes for better efficiency during development.

On top of the efficiency, the utility will read through all your script files and check all the resources you've referenced, and when it is done it produces cleaned-up files, to decrease file size. Decreasing file size is vital when packaging a game for distribution so that the game takes up the least amount of space as possible, whether on a web server, hard drive, or disc. The TGB Packaging Utility maximizes the most out of your space.

Packaging Your Game Using the Packaging Utility

Here's how it works:

1. In the TGB Level Builder, load one of your saved projects.
2. Go to File > Build Project on the main menu. This opens the Packaging Utility dialog box.
3. Set your Output Directory Name. This is the folder your packaged game build will export to. It will be placed in your games directory. Make sure it does not conflict with other folders you might have there.
4. Choose your platform you want to export to. It's a good idea to pick the same platform you've built the game on.
5. Click the Options button to bring up optional configurations, and choose the option to add and/or remove ignored file extensions and directories.
6. Click the Analyze Game button and wait as the utility analyzes and accumulates each file. Once done, a screen will pop up listing everything found for your project.
7. Double-check the list to make sure that all the files you wanted ignored are, in fact, ignored.
8. Click the Build button and wait as the utility packages your game for you. You're done!

Once you are done, you can burn your packaged game to disc, zip it, and put it on a Web server for download, or send it to your friends to play.

Burning Your Game to Disc

If you plan to burn the game to disc, you will need third-party software such as the following:

- **Nero** (http://www.nero.com)
- **Roxio** (http://www.roxio.com)
- **CDBurnerXP** (http://www.cdburnerxp.se)
- **Cheetah Burner** (http://www.cheetahburner.com)

Be sure to read all the materials that come with the software, because each application runs different from the next. There are overlapping similarities and the end results are often alike, but the interface for each can be different and confusing at first. Note that while most CD-burning applications operate the same, you get what you pay for. The cheaper ones have less options and features, while the ones that cost more often give you more control over your finished product.

Creating an Autorun File

If you have created the contents for your CD and then want them to autoplay when the disc is loaded into a computer, you will have to add an autorun file to your CD. An *autorun file* is a simple text file that tells the computer which executable file to start upon loading the CD.

Autorun files are purely optional, as some developers find them annoying and/or time-consuming devices. The proceeding exercise shows you how to write your own autorun files for use with Microsoft Windows operating systems.

For Linux or Macintosh platforms, you will have a more difficult time programming autorun files, and it is suggested that you either do an extensive web search for a how-to or that you compensate by getting yourself a third-party autorun creation utility. Roxio has built-in capabilities for burning hybrid-platform discs, and Toast (http://www.roxio.com/en/products/toast/index.jhtml) is a CD authoring and media conversion software application for Mac OS X that you can use.

There are two types of Windows autorun files, one that uses the `open` command and one that uses the `shellexec` command.

Using the Open Command The *`open` command* is compatible with any PC running Windows 95 or later. However, it is easy to manually disable the autorun features on any computer, so if you hear of someone who cannot view your autorun application, this may be the reason. The `open` command, unfortunately, can only be used for opening executable files that end in .exe. It cannot be used for opening any other document.

An example of an autorun file using the `open` command is as follows. Open your default text editor and in a new text document type:

```
[autorun]
open=myfile.exe
```

That's it! Save as autorun.inf. The .inf extension tells the operating system this is a file to read at startup, or whenever the disc is first put into the computer. The file this autorun is loading during initialization is called myfile.exe, which is a dummy file since we don't really have one.

Using the Shellexecute Command The *`shellexecute` command* is more versatile than the `open` command and will allow you to open nearly any file in its native application. It can be used for opening PDFs, Word files, web pages, and QuickTime movies where the `open` command would not work. This function was only introduced by Microsoft in Windows 2000, so it will only work in Windows 2000, Windows Me, Windows XP, Windows Server 2003, and later.

An example of a `shellexecute` command is as follows. Open your default text editor and in a new text document type:

```
[autorun]
shellexecute=myfile.exe
```

Save as autorun.inf. Notice we're using the dummy myfile.exe file again, and all that has really changed is the `open` command has been replaced by the `shellexecute` command.

A last, but very important, note is that the autorun.inf file should always be placed in the top level of the CD and not placed within a folder.

Physical Packaging

Most blank CDs you buy at the store come with the options of sleeves or disc cases. You can also buy bulk disc cases from online vendors. A really sweet way to dress up your game once you've burnt it to disc is to put it in a disc case and print a custom label for it with artwork or a screenshot from your game right on it! Don't forget to make the title of your game large and noticeable, because you're going to want people to notice and talk about your game right away.

Learn to Advertise Yourself

Whether going to a game publisher or planning to self-publish your game online, you'll find it can be difficult to promote yourself. If you find yourself in this category, you're not alone. Even the singing legend Madonna was prone to self-doubts, and she has been recognized the world over as one of the premiere self-promoters.

You might be plagued by insecure feelings and doubts, or you might be really self-confident but feel selfish or like you have to be humble. Don't be! If you want people to notice you and play your game, you can't be a fly on the wall. You have to be just as crazy, outlandish, and noticeable as possible.

Now I'm not advocating that you dress like Marilyn Manson when you stroll down the streets of your hometown. And if you plan to visit with a publisher or conduct yourself in a business atmosphere, you better put on a nice outfit.

You're expected to dress the part of the environment you're entering; as the adage goes, "When in Rome, do as the Romans do." If you're hanging around a game design company where everyone's wearing T-shirts and blue jeans, you'd better wear a T-shirt and blue jeans as well. But if you're going into a boardroom to negotiate contracts with your publisher, you'd best put on whatever's professionally acceptable for your gender and zip code!

What I am telling you to do is to stop being indistinct, colorless, or wishy-washy. People won't notice you if you don't want to be noticed, and that's a shame because you deserve to be noticed. Now you might be about to say, "But it's my game that I want people to notice, not me!"—and that's true; but if you are an unnoticeable person used to escaping comment and keeping to yourself, then there's a great possibility your game's never going to get noticed either, and that's a crying shame.

Learn to Advertise Your Game

Most game companies have PR and marketing people to cover this angle for the company, but when you start promoting your game, you will be the person singly responsible for getting it noticed. This means you will have to deal with advertisement graphics, promotional pieces, flyers, and other advertisements. Having skills in using an image-editing program and word processing program, English syntax and grammar, and human psychology will help.

Creating a gimmick will also put your product over the edge.

Develop a Clear Identity or Gimmick

A lot of companies talk about using a gimmick. A *gimmick* is a clear and representable image of an idea. A gimmick helps to sell products. You've more than likely seen gimmicks all around you, and some of them can be very transparent or clumsy, but most often a gimmick is purely a clear and representable image of an idea that takes too long to explain.

Think about your game idea. Can you express your game idea in a single, clear sentence? Can you express it in just one image or avatar? If not, you might have to develop a gimmick to sell your game.

Taking time to write down your game helps you sharpen and clarify your game idea to yourself and to the team that you'll be working with. If you skipped doing this or made the game first before taking the time to simplify your concept, then you'll need to work it out right now, before going any further with promotion.

Once you've simplified your game idea, consider the following, and find one thing that would serve as a possible gimmick:

- **Character:** Do you have a cute, sexy, strong, or mysterious character that is dissimilar enough and exciting enough to serve as an icon for your game?
- **Place**: Is the setting for your game adventuresome, glorious, beautiful, or mysterious enough to serve as an icon for your game?
- **Weapon**: Does your character wield an interesting, powerful, cool, or different sort of weapon that looks neat enough to serve as an icon for your game?
- **Enemy:** Do you have a scary, awesome, powerful, or mysterious enemy that is enthralling enough to serve as an icon for your game?
- **Random Element:** Is there some gameplay element so infusive that it's found everywhere in your game and looks different and exciting enough to serve as an icon for your game?

Let me give you just a few examples of gimmicks other games have used, in case you're still confused:

- In *Tomb Raider*, the gimmick is Lara Croft; in *Super Mario Bros.,* it's Mario; in *Donkey Kong Country*, it's Donkey Kong; and in Zelda it's the elf Link. All of these games have a gimmick that's the main character.
- In *Prince of Persia: The Sands of Time*, the gimmick is a combination of the main character (the Prince) and the place (the romanticized Ancient Persia).
- In the games *Diablo* and *Rayman: Raving Rabbids*, the gimmicks are the enemies.

You have to find just the right look for yourself, the right gimmick for your game, and want people to notice you and play your game. Because you've worked hard at creating a fun game, which people will like, you know that you're not offering them empty promises and instead giving them excellent entertainment.

Okay, but how do you get the world to see you and your game? How do you clue folks in that you have something for them to play? One of the best ways in this beautiful cyber age is through online communities.

Pitch a Game Proposal to a Publisher

Most amateur game designers are satisfied working with a close team of their own personal friends, finding a team member with strengths in each facet of the design project to strengthen the whole. Games can be designed on the cheap using Torque Game Builder and self-published online. However, if you feel you could do better with a bigger budget or bigger team of developers, you should consider developing a game proposal and seeking out game publishers to back you.

When working on your game proposal, which is similar if not based solely on your game design documentation, keep in mind the difference between those materials intended for internal use and those you want a potential publisher to see. When writing game proposals, most companies do not include every detail. The most important details the publishers need to see are as follows:

- You obviously know what you're doing and have the appropriate skills to pull off the project you propose.
- The game you propose looks good. Notice I said "looks." You can talk a good game, but until you have a demo (especially a playable prototype) to show a publisher, they generally won't give you the time of day.
- Your proposed game has all the earmarked tags of a Triple-A title (including but not limited to: good graphics, original storyline and characters, big build-up, and unique gameplay).
- You have set your company above all the rest by being fresh, innovative, and appealing. You have to have a clear identity and a great gimmick (see the following section for more information).

When you go before a publisher or design house, try to secure a face-to-face meeting with them to deliver and review your proposal. This way, you can

elaborate on specific points you think are important and you can answer clarifying questions. Just remember to go in prepared, collected, and dressed right (good hygiene is important!).

Your proposal can be on paper, but it is recommended to use visual media. Something that is overused but very popular for presentations is putting together a presentation slideshow using Microsoft PowerPoint. You could also set up a playable game demo right there, or show them a prerecorded demo of gameplay.

What you really want is the freedom to discuss and evolve your game description while answering the publisher's questions. The energy you impart to the publisher will equal their willingness to listen, in most cases, so go in pumped and excited about your own game.

Usually, if you're under the age of 18, a publisher will want to work with someone older as an intermediary, if they give you the time of day at all. Be forewarned this is not an easy row to hoe; in fact, it's much easier to self-publish and self-promote your own games, which is an easy plan to choose in this cyber age of garage games.

Advertise Your Game Online

The Internet is a global network of computers that enables people around the world to share information. Whereas the word *intranet* denotes a private network, the word *Internet*—with a capital I—is a global network of networks, and it is growing on a daily basis.

Note

Why, you might be wondering, was the Internet created in the first place? When the Soviets launched Sputnik, the first space satellite, in 1957, the U.S. government responded by instigating an aggressive campaign to stay ahead of its global competitor; hence, the Advanced Research Projects Agency (ARPA), the U.S. agency in charge of space and strategic missile research, was born. ARPA created the first indestructible computer network, connecting computers, then housed in universities and government agencies, to prevent the destruction of important data in the event of an attack on the United States. (As an aside, the computers then in existence were giant mainframes—the kind that took up entire rooms. The small personal computer that is ubiquitous today would not arrive for several years.) This network served as a precursor to ARPANET, the first glimmer of what would become the Internet, which was born in 1969.

Although users around the world have different types of computers and operating systems (OSes), the Internet enables them to share information through the

use of protocols. *Protocols* are common sets of rules that determine how computers communicate with each other over networks. For instance, each computer that is connected to the Internet connects using Transmission Control Protocol/Internet Protocol (TCP/IP).

What Is the Web?

Many people use the terms interchangeably, but the Internet and the World Wide Web are not one and the same. The *World Wide Web* (or WWW), often referred to as simply the web, is actually a subset of the Internet that supports *web pages*, or specially formatted documents created using languages such as *Hypertext Markup Language* (HTML). Hypertext allows you to click words in a document that are linked to related words, graphics, or other elements in the same or in another document. Put another way, you might think of the Internet as the connection between various computers worldwide, and the World Wide Web as the content that resides on those computers that is transmitted via these connections.

Getting an Internet Connection

In order to maintain your own website, you'll of course need a computer, and you'll need that computer to be connected to the Internet. That means you'll need an Internet service provider (ISP). These providers offer connections of varying types, including the following:

- Dial-up, using a telephone line and modem
- Digital Subscriber Line (DSL), using a telephone line and DSL modem
- Cable, available anywhere cable TV is offered
- Fixed wireless, using either satellite or microwave technologies
- Mobile wireless, using cell phone or wireless fidelity (Wi-Fi) technologies

Often your personal circumstances will dictate which option is best for you. Before picking an Internet connection, you must do your homework and check speed as well as pricing.

If you don't have your own personal computer with access to the Internet, you can use the Internet connections provided for public access in places like libraries, colleges, and universities. Although certain restrictions may apply, computers in these places are typically available for use by the general public.

Even if you do have your own computer with Internet access, it's nice to know these public computers exist. For example, you might use them to update your web pages if your home machine is out of commission.

Web Development

Think of your website as a container. In that container, you'll need to put *stuff*, or content. You'll want to have a good idea of what kind of stuff, as well as how much stuff, you'll be putting in your container before you build the container; that way, you'll know what kind of container you need and how big it should be. After all, just as you wouldn't put glass marbles in a metal pan to carry them a long way, and you wouldn't put feathers in a fishnet sack, you won't want to create an enormous site for just a bit of content or vice versa. So before you get carried away with "I want! I want!" consider your content and your message to determine what you really *need*.

Does your website contain a lot of text? Is it going to be mostly images? Does it contain an equal mixture of the two? How often do you think you will be updating your site? Daily? Weekly? Monthly? Never? Answering questions like these will help you determine what kind of website you need. If you plan to build the site and never touch it again, a static site that looks really good but might be harder to edit is right for you. If you want to update it daily or weekly, you may need a blog-style site that may appear plain but can be edited in a matter of minutes. Of course, given the right tools and know-how, you can find the compromise that works best for your situation.

The following information gives you a brief primer for web development. This subject is so vast, however, that it could take up several books. If you decide you really want to publish your games on the web, there are whole books dedicated to that topic that you should read. A few I'd suggest from Course PTR include:

- *Web Design for Teens* by Maneesh Sethi
- *Principles of Web Design, 4th Edition*, by Joel Sklar
- *Web Design BASICS* by Todd Stubbs and Karl Barksdale
- *PHP for Teens* by Maneesh Sethi

Prepping Your Images Chances are, your site will make frequent use of images. Note that these images must be small enough for transmission over the Internet. (When I say "small," I'm referring to the size of the image file,

not the dimensions of the image itself.) Unlike images you prepare for print, which must have a resolution of 150 to 300 dots per inch (dpi), an image bound for the web needs to have a resolution of 72 dpi with a file size of 200 kilobytes (KB) or less. In order to achieve this file size, you will likely need to compress your images; this reduces redundancy in image data, often without a noticeable loss in image quality. Compressing your images not only makes it more convenient for you to upload them to the web, it also benefits your visitors because it enables your website to load more quickly, displaying your graphics almost as soon as the text appears.

There are three types of widely supported images on the web:

- **JPEG:** JPEG (pronounced *JAY-pehg*), short for Joint Photographic Experts Group, is among the most common image-compression formats available.
- **GIF:** The Graphics Interchange Format (GIF) is an eight-bit-per-pixel bitmap format that supports alpha transparencies. You can pronounce GIF with a hard or soft "G" sound.
- **PNG:** Short for Portable Network Graphics, the PNG (pronounced *ping*) format is a bitmapped image system with 24-bit RGB colors and improvements over GIF. PNG is great for creating low-sized files with excellent quality.

The differences between these three image types are marginal. Typically, game designers just choose the one that suits their work the best.

Prepping Your Text Content Before you build your site or update it with new material, take time to write all your web text beforehand. Your best bet is to use a word-processing program; I like Microsoft Word. It enables you to check your spelling and it even makes suggestions relating to grammar and usage. You can right-click words to view your options, including other spellings, word choices, and synonyms. This both simplifies the process of writing and double-checking your work and speeds it up. In addition, writing things out beforehand helps you ensure that you're conveying the message you want on your web page without being distracted by any coding or technical aspects that are sure to crop up later.

Web Browsers and Web Servers A *web browser* is a software program that is used to locate and display web pages. More than likely you've heard about

or even used some of the most popular web browsers, like Internet Explorer, Mozilla Firefox, and Netscape Navigator. All browsers can display graphics in addition to text. Additionally, they can display sound and video, although many require plug-ins in order for these features to work correctly. For instance, you may have noticed on sites like http://www.newgrounds.com that a plug-in called Adobe Flash Player is required to display games and videos; you must download this plug-in for your browser to play the media correctly.

It all seems very simple. You click, and a new page appears on your screen. But where do these pages live while they're not being looked at? Where are sites stored? Websites and the pages they contain are stored on special computers called web servers. A *web server* is a computer that is hooked up to the Internet 24/7. Web servers might have one or more websites stored on it at any given time; the number of sites and pages that can reside on a server depends on the server's memory capacity. When you enter a web page address in the Address bar of your web browser, the web server responds by sending a copy of that page to your browser.

Addresses and Domains In order for a browser to locate web pages, it must know the page's IP address, domain name, and URL.

An *Internet Protocol (IP) address* is a number that uniquely identifies any machine connected to the Internet, including computers, printers, and mobile devices. IP addresses can be static or dynamic. If you log in to the Internet via a broadband connection, you probably have a static IP address—one that never changes. But if you log on by way of dial-up, your ISP most likely assigns you a temporary or dynamic IP address for the amount of time you are connected.

IP addresses consist of four groups of numbers separated by periods, such as 168.212.226.20. Obviously, you can't be expected to remember a series of digits like this. For this reason, domain names were created. A *domain name* is a text alias for one or more IP addresses. An example of a domain name is microsoft.com; you type this into your web browser instead of the IP address for Microsoft's web server.

According to Weboepedia, "because the Internet is based on IP addresses, not domain names, every web server requires a Domain Name System (DNS) server to translate domain names into IP addresses." A *Domain Name System (DNS) server* is separate from the web server; therefore, when you put up a new site on the World Wide Web, you must find both a web server to host your web pages

and a DNS service to host your domain name. This ensures that the domain name is resolved to an IP address directed at your web server. You'll learn how to obtain a domain name and a web host in a moment.

The URL Each web page—even a page deep within a site—has its own unique uniform resource locator (URL) address. This URL is composed of the protocol used to access the page (for example, http), the domain, and possibly the folder in which the page is stored on the web server. To get a handle on this, take a look at the anatomy of a URL. Following is the URL of a web page:

```
http://www.garagegames.com/products/torque/t360/
```

- The http:// portion of the URL is the protocol. Another protocol you might see is ftp:// used for file transferring.
- The www portion indicates that the page resides on the World Wide Web.
- The domain name, garagegames.com, comes next. The .com tells you that this is either a business-related or private site.
- The third part of this URL indicates the folder in which this page is stored. Because this page uses XHTML coding, you don't see the page name, but the page typically appears with an ending like .htm or .html if it was designed with Hypertext Markup Language, or with .xml if Extensible Markup Language was used.

Building and Maintaining Your Website Adobe Dreamweaver (see Figure 9.1) is the premier website-construction kit for professionals. It allows pros to work in either a WYSIWYG (what-you-see-is-what-you-get) or code environment—or in both simultaneously. Dreamweaver, created primarily for users who have purchased server space online and own their own domain name, comes with several built-in site templates; all you need to do is add your content and create your custom logo. In addition, there are many free templates available online.

Note that Dreamweaver can be fairly complicated; if you decide to use it to build your site but are unfamiliar with the software, you will want to have handy some resources to help you navigate the program. I would suggest, if you are using CS3, to read *Adobe Dreamweaver CS3 Revealed* by Sherry Bishop from Cengage Delmar Learning, 2007. You can learn more about Dreamweaver on Adobe's site at http://www.adobe.com/products/dreamweaver/.

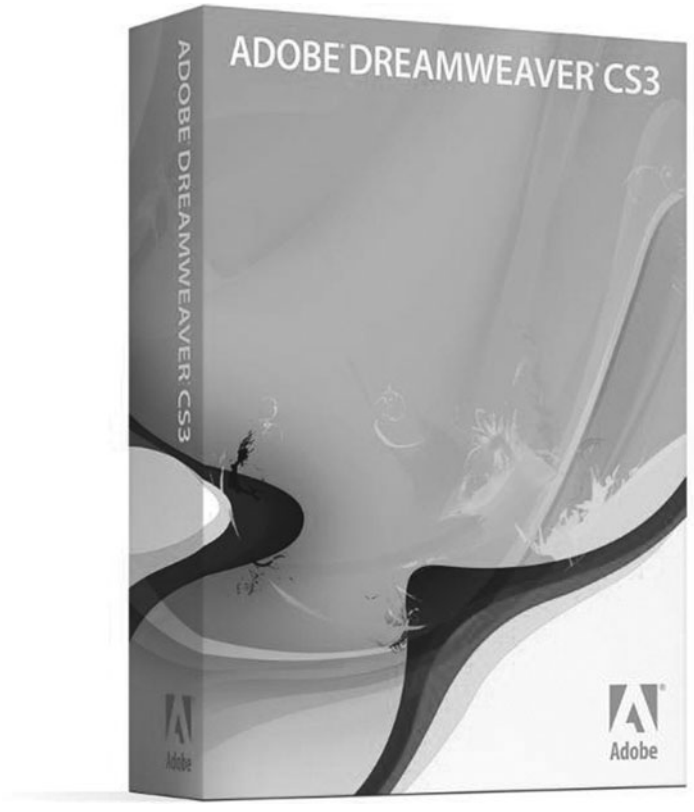


Figure 9.1
Adobe Dreamweaver CS3 box art.

FrontPage, from Microsoft, used to be a popular web-authoring software beginning designers used, and you may have heard designers mention it before, but it is now being rapidly replaced by Microsoft with their new application Microsoft Expression. Expression's goal is to replace the predominance of Dreamweaver and the limitations of FrontPage with a more viable alternative. Expression is most compatible with the Windows OS, obviously, including Windows XP and Vista, and there is a growing community of Expression users. You can learn more about Expression on Microsoft's site at http://www.microsoft.com/expression/.

Another alternative, especially if you don't have the budget to afford Dreamweaver or Expression, is Nvu (pronounced "N-view"), which you can find online at http://www.nvu.com. Nvu is an open-source web-authoring application for Windows, Mac, and Linux users, and this free program provides a great WYSIWYG editing environment and built-in file transfer system to satisfy most

designers' needs. If you've always wanted to get your feet wet building websites, you might try Nvu.

Note

Cascading style sheets (CSS) is a computer language used to describe the presentation of structured documents that declares how a document written in a markup language such as HTML or XHTML (Extensible Markup Language) should look. CSS is used to identify colors, fonts, layout, and other notable aspects of web document staging. It is designed to facilitate the division of content and presentation of that content so that you can actually swap out different looks without changing the content. CSS is thus separate from the HTML/XHTML page coding. CSS is supported in Adobe Dreamweaver, but it can be hand-coded in Notepad or BBEdit in the same manner as the markup language.

Getting a Domain Name As mentioned, domain names are text aliases for IP addresses, making it easier for people to remember how to access a particular web page. When you're ready to build your website, one of your first steps will be to obtain a domain name for it. To get a domain name, you use an online domain name service. Some more popular services at the time of this writing are as follows:

- **Dotster** (http://www.dotster.com)
- **GoDaddy** (http://www.godaddy.com)
- **MyDomain.com** (http://www.mydomain.com)
- **Register.com** (http://www.register.com)

Getting a Web Host In addition to obtaining a domain name for your website, you will also need to obtain an account with a web host. There are countless choices; some are free, and others cost. Following is a list of free web-host services:

- **110MB** (http://110mb.com)
- **50Webs** (http://www.50webs.com)
- **AtSpace** (http://www.atspace.com)
- **Byethost** (http://www.byethost.com)
- **FreeHostia** (http://freehostia.com/free_hosting.html)
- **FreeWebs** (http://freewebs.com)
- **GeoCities** (http://geocities.com)

- **Jumpline** (http://www.jumpline.com)
- **Tripod** (http://www.tripod.com)

When choosing free hosting, go with a reputable host. Some free hosting sites add bulky code to your page, which increases the loading time of your page. Others place advertisements on your page or even program code that can download scripts to your visitor's computer, infecting them with spyware. *Spyware* is not a virus, but it can be just as destructive, loading the infected computer down with random pop-ups, changing the start page on the browser, and communicating to a third party about the user's computer habits. Avoid these types of hosts if you can.

Companies with dedicated web servers will cost you, but as in everything else in life, you get what you pay for. The top web hosts at the time of this writing are listed here:

- **BlueHost** (http://www.bluehost.com)
- **Dot5 Hosting** (http://www.dot5hosting.com/dot5/index.bml)
- **Hosting Department** (http://www.hostingdepartment.com)
- **HostMonster** (http://www.hostmonster.com)
- **HostPapa** (http://hostpapa.com)
- **InMotion Hosting** (http://www.inmotionhosting.com)
- **StartLogic** (http://www.startlogic.com)

Design Dos and Don'ts Here are the most important tips I can give you to help you design your very best web pages:

- When you surf pages online, the items at the very top of a page and off to the left and right are usually noticed the least. Most web pages have advertisements, navigation links, or other information there; as a result, due to our extensive web use, our brains have become programmed to ignore those areas on web pages. Don't put images and content there that you want to be seen; instead, put them dead center in the middle of the page.
- Don't use clashing fonts and colors in your text. A sure sign of amateur web design is when the text contains multiple fonts—especially a mix

of serif and sans-serif fonts—and bright, clashing colors like pink, yellow, or blue. To give your readers a sense of simplicity, cohesion, and artfulness when they come to your website, use one font throughout your web pages, and make sure to use harmonious colors.

- Employ a clean, interconnected design that blends well throughout all the pages on your site. You don't want your visitors thinking they've left your site when they click a link to one of your interior pages and it doesn't mesh with the page they just left; they might shrug and go away—and you don't want that!
- Don't use competing graphics. If your artwork—your screenshots or concept artwork—is struggling to be seen because it's buried under opposing images and too-powerful secondary art, you will lose your audience because visitors won't know what they're supposed to look at first.

Uploading Your Files When you obtain your domain name and web host, keep *everything* organized—all passwords, IP addresses, host names, and anything else connected to your website. Write it all down in a notebook and put it somewhere safe so that you don't lose it. Otherwise, you might well lose access to your site and have to perform some complicated operation to restore access.

You'll need all this information whenever you edit your pages or upload your files to the web via FTP. *File Transfer Protocol* (FTP) allows you to upload and download files to and from the server. If you have Dreamweaver, you can use FTP from within the program to communicate with the server. Other programs, such as SmartFTP or CuteFTP, can be used to upload files you code by hand.

Getting Your Site Noticed Now that you've gotten some real estate in cyberspace and you're confident in your overall design abilities—that is, you believe people will enjoy visiting your site and downloading your games—then it's time to open the doors wide and let people in. Your first step is to submit your new site to search engines so that it can be indexed.

Pages are published to the World Wide Web by their domain owners or by contributors—and just as easily, they can be changed or taken off the site. Thus, a page may be there one week and gone the next. This makes the web an ever-shifting environment. Humans can compile directories of web links that point to subjects of interest, but if these same people don't check up on their links, they may quickly become dead ends—and nothing spells a dusty site like a bunch of

dead links. This is why we've developed other methods for searching the World Wide Web for the content we want: search engines and metasearch engines.

Search Engine Optimization If you know exactly what website you want to visit, you can simply type its URL into your browser's Address bar. If not—if, for example, you know the information you seek is out there somewhere, but you're not sure where to find it—you can conduct an online search. Recent years have seen a proliferation of search tools, including the following:

- **Search engine:** A *search engine* is a program that searches the web for specific keywords and returns a list of documents in which those keywords are found. Popular search engines include Google, MSN, Yahoo, AOL, Alta Vista, Dogpile, and Ask.com. A search engine is composed of three parts: a *spider* or *crawler*, which "crawls" through all the websites for keyword traces; an index of sites containing those keywords; and a search algorithm that makes it all happen. How the results are arranged and ranked varies by engine, although most try to put what they determine to be the most relevant and authentic results near the top of the list.

- **Metasearch engine:** *Metasearch engines* like Vivisimo and Metacrawler do not actually crawl the web to build their listings. Instead, they send the search info to multiple search engines at once and then compile their results. This can speed up your searches.

Following is a list of search engines to which you should consider submitting your site:

- **AOL Search.** Locate the appropriate category and follow the onscreen instructions to submit your site.

- **AltaVista.** Visit http.//addurl.altavista.com/sites/addurl/newurl and follow the onscreen instructions.

- **Ask.com.** E-mail your site's URL, along with a description, to url@askjeeves.com.

- **Google.** Visit http.//www.google.com/addurl.html and follow the onscreen instructions.

- **Hotbot.** Visit http.//hotbot.lycos.com/addurl.asp and follow the onscreen instructions.

- **Lycos.** Visit http.//www.lycos.com/addasite.html or http.//www.alltheweb.com/add_url.php and follow the onscreen instructions.
- **MSN Live Search.** Visit http://search.msn.com/docs/submit.aspx.
- **Open Directory Project (ODP).** Locate the appropriate category and follow the onscreen instructions to submit your site.
- **Yahoo!.** Locate the appropriate category and follow the onscreen instructions to submit your site.

In addition to submitting your site to search engines, it's also a good idea to employ search engine optimization (SEO) techniques. *Search-engine optimization* (SEO) is the method by which web designers make websites more visible to search engines, therefore getting more people to come to their web pages.

Meta Content Although each search engine employs a different search algorithm, most are partial to scripted HTML meta content—namely descriptions, unique page titles, keywords, and long-tail keywords. For an example, take the following:

```
<head>
 <meta name="KEYWORDS" content="dragons sorcery sword orcs elves dwarfs role-
playing game gaming interactive fantasy">
 <title>Lopping Heads: a Fantasy Online Game</title>
</head>
```

Long-tail keywords are entire groupings that you'd see used in a search phrase such as "coolest fantasy online game," while *keywords* are one-word descriptors like "role-playing."

Keywords used in the meta content should also appear in the body content, and justifiably so. If you describe your site as being about "dragons" and the text "dragons" never shows up anywhere in your writing, then you've committed false advertisement or at the very least decreased the chances that someone who will be interested in your content has found your site.

Links and Link-Backs Another important factor of SEO is linking. Most search engines grade site performance based on how many strong web links there are going out of and coming into the site. If you place a web link to Microsoft—and Microsoft links back to you—your chances at appearing in the top 20 on a search engine go way up. Find like-minded game design

groups who are willing to trade links, and link to some of the larger amateur game indexes while getting your link added to their site.

Using Blogs or Online Communities to Get Noticed

Besides building and uploading a custom website to a web server, you could consider other, less time-consuming methods of getting your content on the World Wide Web. The two most-used methods are blogs or online communities.

Blogs

Blogs (short for web logs) are websites maintained by a blogging service or individual with fairly regular entries of observations or other material such as graphics or video. Entries are commonly displayed in reverse-chronological order or most recent post first. Many blogs provide commentary or current events on a particular subject, while others function as more personal online journals. ''Blog'' can also be a verb, meaning to make and maintain a web log.

Some popular free blog services include:

- **Blogger/Blogspot** (http://www.blogger.com)
- **i3log.com** (http://www.i3log.com)
- **Windows Live Spaces** (http://home.services.spaces.live.com)
- **Wordpress.com** (http://www.wordpress.com)
- **Yahoo 360** (http://360.yahoo.com)

Online Communities

Online communities are websites designed to foster communication and networking. They've been compared to bulletin boards, social clubs, and schoolyards. They're somewhere you go to chat and interact with ''real world'' peers—as well as people you've never met before, from all over the world.

The World Wide Web, since its inception by Sir Tim Berners-Lee, has exploded with hundreds of community websites, many of them utterly vast in scope, such as MySpace, Friendster, Bebo, and Facebook.

Most social networks on the Internet are open to the public and anyone can join, although each individual site has its own User Agreement which may have

specific rules or regulations joiners must follow. Many of these networks are free to join, which is one of the reasons they're so popular. Sites like MySpace are exclusively funded by the advertisements taking up screen real estate. Another plus to these networks is you don't even have to use your real name. You can viably create unlimited alter egos or work as an anonymous entity.

What you get out of these sites is entirely up to you, of course. But as we are discussing the promotion of your games, that's one thing you can achieve.

Setting Up a GGE Account

The Great Games Experiment, or GGE, hosted by the folks at GarageGames, is an online community with over 20,000 registered users that's sole purpose is for gamers to find games and game developers to share their games and game design advice. GGE is one community you can join to promote your games to the world. You can find it online at http://www.greatgamesexperiment.com.

When you first load the homepage of GGE (as seen in Figure 9.2), you should see a button in the top-right corner that says Create Account. Click it to come to the free account creation page, where you will enter an e-mail address and an activation message will be automatically sent to that e-mail address. The activation message can take up to 10 minutes to reach your e-mail inbox. When you find it, follow the activation link to finish setup of your account.

Under your account setup, you can choose to be a Gamer or a Developer; as you're trying to get people to play your games, you should choose Developer.

You will be given different options as a Developer, including Developer Skills that you rank from Novice to Advanced. Come up with a short description of your game creation company, an avatar that befits your game vision, and a Gamer Tag. The Gamer Tag is similar to a domain name, as it will help gamers find you on GGE. GGE also offers a specialized coding scheme for your text field inputs, called GGE code. You can view a short list of GGE code formatting by clicking the GGE code button above the input field under Describe Yourself.

Once you have a GGE account created, you can use it to advertise your games, share trailers for it, have people download demos, and network with both gamers and other developers.

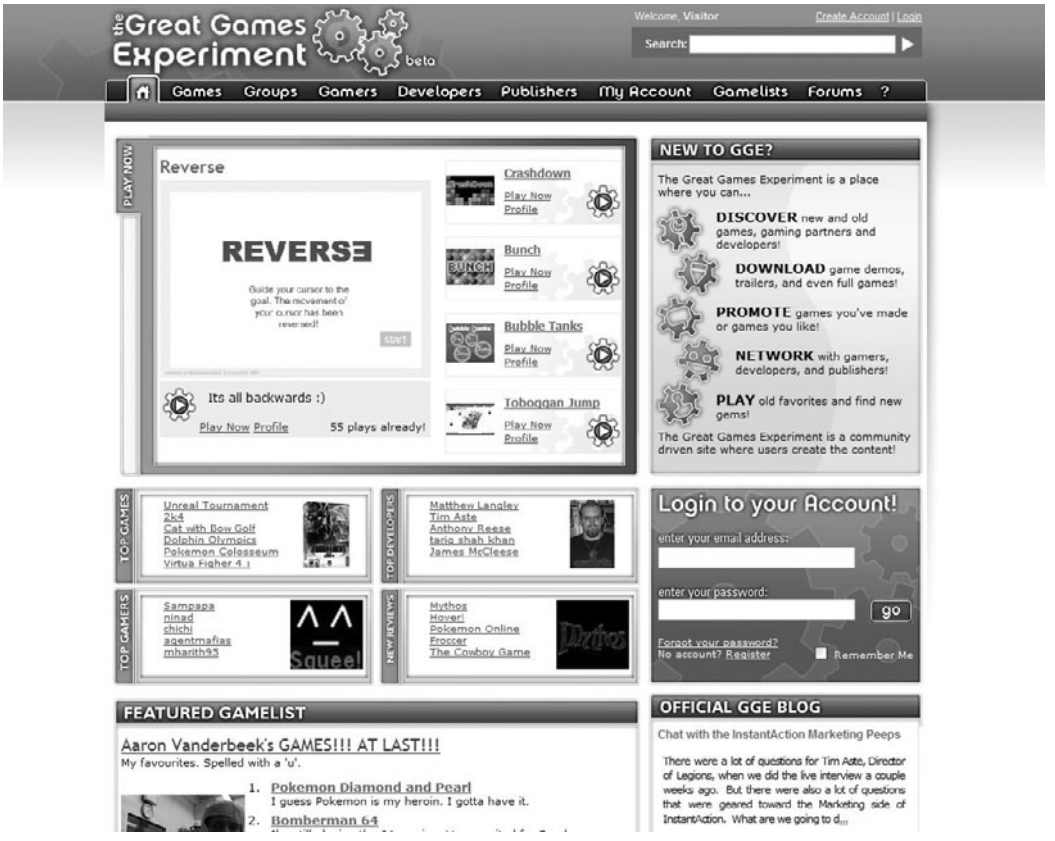


Figure 9.2
The Great Games Experiment welcome page.

Setting Up a MySpace Account

With over 200,000 new accounts created each day and over 106 million users, MySpace (seen in Figure 9.3) is one of the largest online communities and could net you thousands of players and friends. It is a proven ground for sharing and experimentation. You are free to join as many online communities as you want and promote yourself and your games on any of them that allow it, but for now, let's focus on getting into MySpace.

If you're some kind of Internet-savvy teenager, there's a very big chance you already have a MySpace page. If this is the case, feel free to skip this section entirely. You can begin promoting yourself right away by placing some images from your game, some Desktop goodies, and playing a short trailer on your site pages. Then invite all your online friends to try your game and spread the word

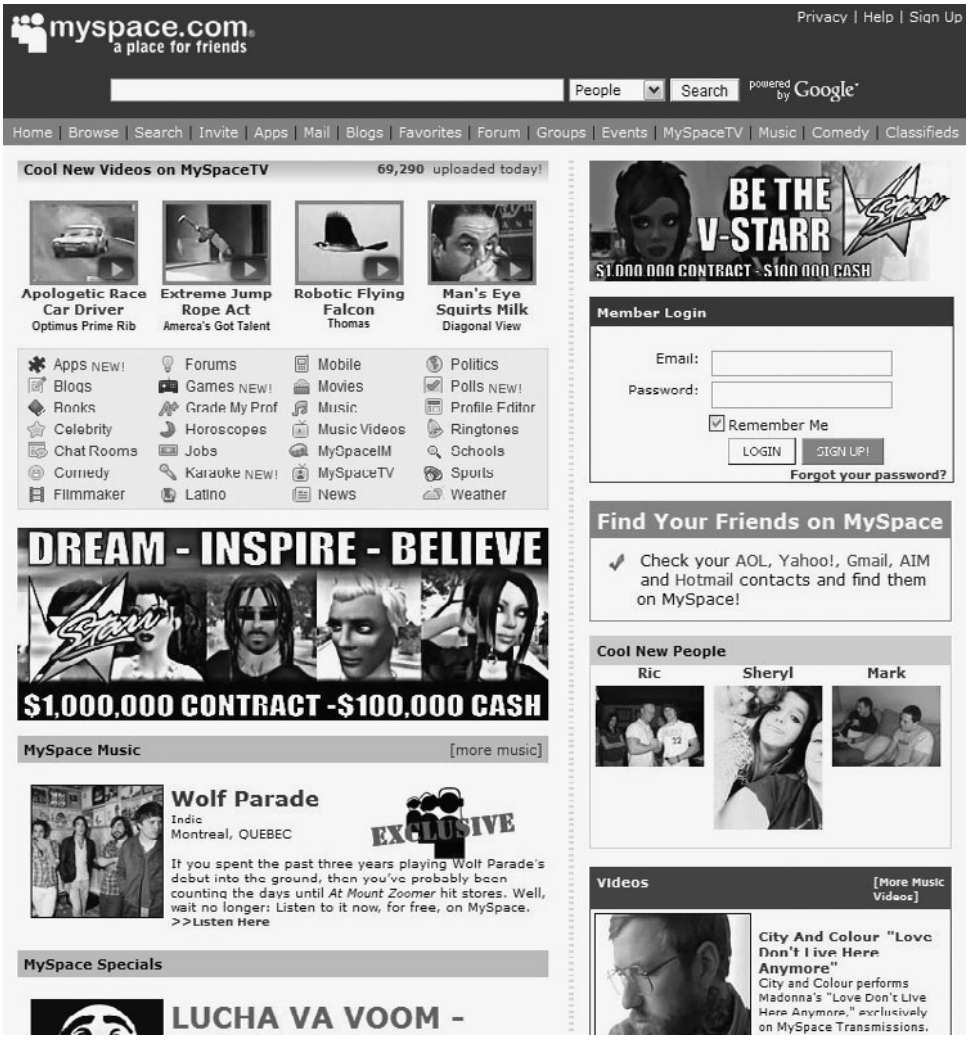


Figure 9.3
The MySpace welcome page.

through chat. If you've never considered a community account before and don't even know what MySpace is, then you should read on.

Go to http://www.myspace.com and you should find yourself on a local MySpace site. Click the Sign Up button in the top-right corner of the page and fill out the online form that appears. By default, your Profile page will only publicly display your first name, age, and location, so don't worry about your privacy at this time. You have to be 14 years or older to sign up for a MySpace account, however.

MySpace has begun cracking down on underage account holders, so you might have to try Facebook or Friendster instead if you're younger than 14.

You have to have an active e-mail address or web mail account. After you click on Sign Up at the end of filling out this form, you'll be prompted to upload your first image, which will appear as your face on MySpace until you change it. You can't submit anything offensive or rude or the copyrighted work of others. In fact, as a game designer you should use your personal photo, a company logo (if you have one), or some art from your game. You can only upload a JPEG or GIF and it must be smaller than 600 KB in file size.

Next, you'll assign your MySpace URL or name (they're the same thing). The URL, or Uniform Resource Locator, is your profile's address on the web. Your URL at the start will begin (always) with http://www.myspace.com/ and will end in a ridiculous string of numbers. Click on the link and enter your desired moniker; it can be anything as long as nobody else is using it, but keep in mind that it should be short and it should be descriptive.

You want people to associate you with the games that you build. For this, you're going to have to make an online personality. Keep in mind the image you want to portray and the gimmicks for your game that you've developed. To get started, click the Edit Profile option found in the list next to your image at the top of your Hello page. This opens the Profile Edit page. The page's tab reveals the numerous sets of fields where you can add personal details.

The most important field here is the one that says About Me. Tell the whole world who you are, what games you make, and you can write it however you like. You might want to pre-write this text copy in Microsoft Word first, spell-check it, or even run it by your friends or parents, before you add it to your MySpace page. This will develop people's first impression of you, so you want it to matter.

You might want to blog. You could type entries online covering the development cycle of upcoming games you're working on; this is known as keeping a development journal online. Or you could enter updates, and when a new game is coming out or post bug reports. Whatever you can think of to blog, you can do it from your MySpace account. Enter your Blog Control Center and click on Customize Blog in the My Controls box. A page will come up with a long list of tabs and fields, each controlling different page elements and customizing the look of your blog. You can read more about the blog on www.myspaceblogr.com.

There's a community in MySpace for games, although usually it's for gamers looking for games. This is a great place to make friends. However, besides short Flash games and games with relatively small file sizes, you cannot upload your Torque games to MySpace. You'll have to get your file hosted somewhere else and then post a link to your game in your blog or directly on your Profile page (and anywhere else you want to, too).

The following are some places you can find to host your files:

- **Files Upload** (http://www.files-upload.com)
- **Wiki Upload** (http://www.wikiupload.com)
- **MyDataBus** (http://www.mydatabus.com)

What's Next?

The sky's the limit! You now have the beginnings of becoming a great game designer. You know what to do, but especially what not to do. You know how to put together a video game using the Torque Game Builder. You know how to promote yourself and your games.

The next step is up to you. Are you going to sit on the couch and play games? Or are you ready to take it to the next level and sit in front of your computer and make awesome games all your own? You have to make that decision. What's it going to be?

Game Design Schools

If you decide game design will be your career goal, get through high school with good grades, especially in art, math, and science (specific classes you should take include drawing, design, computer science, physics, and geometry). Then jump into a college that teaches game programming or game art; here are a few:

- 3D Training Institute
- Academy of Art University
- American Sentinel University
- Collins College

- Daniel Webster College
- DeVry University
- DigiPen Institute of Technology
- Digital Media Arts College
- Emagination Game Design
- Ex'pression College for Digital Arts
- Full Sail Real World Education
- Global Institute of Technology
- iD Tech Camps
- International Academy of Design and Technology
- ITT Technical Institute
- Media Design School
- Pacific Audio Visual Institute
- Sanford-Brown College—St. Charles
- Seneca College's Animation Arts Centre
- The Academy of Game Entertainment Technology
- The Art Institute Online
- The Florida Interactive Entertainment Academy
- The Game Institute
- The Guildhall at SMU
- The School of Communicating Arts
- University of Advancing Technology
- Vancouver Institute for Media Arts

- Westwood College of Technology
- Westwood Online College

The choices are up to you. The beauty of it is that you're entering a whole new and growing field of entertainment that is quickly surpassing video and books, and with the skills you develop right here and now, you can set yourself up for a future career doing what you love!

Review

After reading this chapter, you should understand the following:

- How to advertise yourself and create a clear identity or gimmick
- How to pitch a game proposal to a publisher
- What the World Wide Web is and how it operates
- Where to get started building and maintaining a web presence
- Getting your website noticed and searched by search engines
- How to set up and use online community accounts such as the Great Games Experiment and MySpace
- What to consider when it comes to future goals and your education

APPENDIX A

Keyboard Shortcuts and Operators

The first part of this appendix is a list of keyboard shortcuts for use in the TGB Level Builder. The second part of this appendix covers the operators and keyboard events to be used within the TorqueScript programming language.

Keyboard Shortcuts

The following is a list of keyboard shortcuts (also called macros) for use with the TGB Level Builder. These can be customized from within the editor, if you so wish. Alternate key commands for Macintosh users are given in parentheses where applicable.

Shortcuts	
Activate the Selection Tool	ESC
Copy Selected Object	CTRL+C (CMD+C)
Cut Selected Object	CTRL+X (CMD+X)
Home the View	HOME
Move Selection One Layer Away from Camera	]
Move Selection One Layer Closer to Camera	[
Move the View Window Down	Down Arrow
Move the View Window Left	Left Arrow
Move the View Window Right	Right Arrow
Move the View Window Up	Up Arrow
Paste Cut or Copied Object	CTRL+V (CMD+V)
Redo and Undone Action	CTRL+Y (CMD+Y)

Show All Scene Contents	END
Show the Create Panel	C
Show the Edit Panel	E
Show the Project Panel	P
Undo Last Action	CTRL+Z (CMD+Z)
Zoom the View Window In	=
Zoom the View Window Out	–

Operators and Keyboard Events to Bind

The following is a list of general operators used frequently in TorqueScript. If you memorize the basics, you will have an easier time reading and writing your own scripts.

Operators	
$	A global variable
%	A local variable
*	A times sign, used for multiplication
/	A division sign, used for dividing
+	A plus sign, used for addition
–	A minus sign, used for subtraction
++	A double plus sign, used for auto-increment
––	A double minus sign, used for auto-decrement
<	Less than
>	More than
<=	Less than or equal to
>=	More than or equal to
==	Equal to
!=	Not equal to
!	Not
&&	And
||	Or
=	Assignment of values
op=	Compound assignment of values
@	Concatenation of multiple strings/variables into one string
$=	String equal to
!$=	String not equal to
?	Conditional logic
[ ]	Array element container
( )	Grouping container
{ }	Blocking container

,	Listing
::	Namespace
" "	String container for a normal string
' '	String container for a tagged string
//	Single line commented out
/* */	Multiple lines commented out

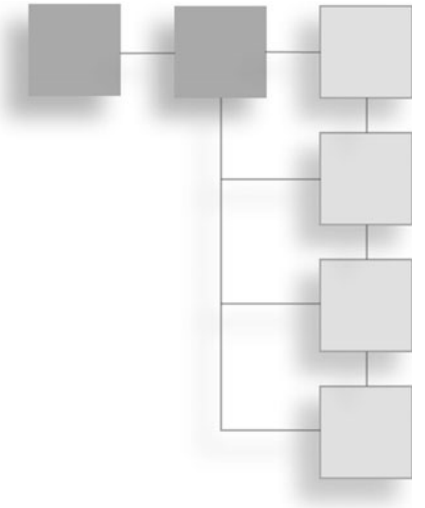
The following is a list of general keyboard events, or how they should be written in the code, to bind.

Keyboard Events to Bind

alt	numpad8
apostrophe	numpad9
backslash	numpadadd
backspace	numpadadmult
capslock	numpaddecimal
cmd	numpaddivide
comma	numpadenter
ctrl	numpadminus
delete	numpadsep
down	opt
end	pagedown
enter	pageup
equals	pause
escape	period
help	print
home	ralt
insert	rbracket
lalt	rcontrol
lbracket	return
lcontrol	right
left	ropt
lessthan	rshift
lopt	scrolllock
minus	semicolon
numlock	shift
numpad0	slash
numpad1	tab
numpad2	tilde
numpad3	up
numpad4	win_apps
numpad5	win_lwindow
numpad6	win_rwindow
numpad7	

This page intentionally left blank

APPENDIX B



What's on the CD?

The following is a list of materials you will find on the companion CD-ROM, all of which can be very helpful to you fulfilling the exercises in this book.

Software Demos

Torque Game Builder version 1.74 (Mac/Win)

Torsion version 1.13 (Win)

Platformer Starter Kit (Win)

Game Demos

King Kong (Win)

Magic Pearls (Win)

Phantasia II (Win)

Pirate Tales (Win)

Puzzle Poker (Mac/Win)

Tank Buster (Win)

Data Files

Chapter 4 Files

Chapter 6 Files

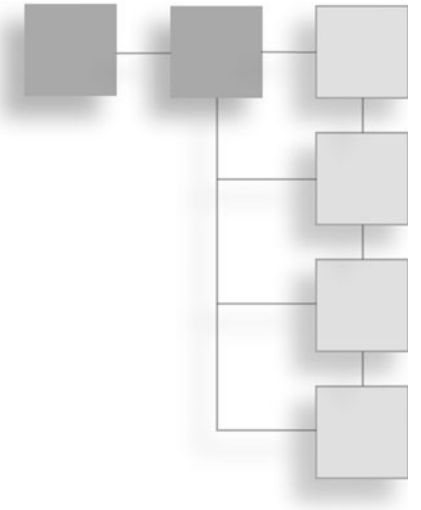
Chapter 7 Files

Chapter 8 Files

Royalty-Free SFX

5 original Mojo Audio sound effects sample MP3s

Appendix C



Glossary

Here you will find the most important terms used within the text of this book and their meanings.

Glossary Terms

2D/3D hybrid games—in these games, the scenes were painted 2D backdrops with an invisible 3D box placed over the top of them so that players couldn't wander out of the scene's confines; the player characters and enemies were composed of 3D polygonal models

3D graphics—virtual realistic scenes created from 3D polygon primitives rather than flat 2D vector or bitmap images; 3D graphics have become a big potential selling point for video games and are quickly dominating the console market

action games—games where the player's reflexes and hand-eye coordination make a difference in whether she wins or loses

adventure games—games that traditionally combine puzzle-solving with storytelling; what pulls the game together is an extended, often twisting narrative, calling for the player to visit different locations and encounter many different characters

algorithm—a set of instructions, listed out step-by-step, to your computer, to make it do what you want it to do

animated sprite—a 2D rectangular image that will be animated, like a character walking across the screen or a waving flag on a pole

art bible—a compilation of all the concept artwork, design notes, and drawings of the characters, vehicles, environments, and weapons for use by the artists making a game

audio compression—a process that restricts the range of sound by attenuating signals exceeding a threshold

avatar—the onscreen representation of the player's character, or the protagonist of the game narrative

beta testing—a method used in game development; after the entire team has approved of their own game, they'll often pull in people not related to the team to test a game with a set of fresh eyes

Bezier curve—vector art curves that let designers create extremely accurate drawings using complex curves with precision handles

bit—the smallest measure of digital data that comes from the phrase "binary digit," and is either a 1 or a 0

bitmaps—images that are fixed-resolution images, generally scanned paintings or drawings, composed of tiny squares of color information called pixels

byte—the next smallest measure of digital data; is composed of eight bits

camera—an often invisible, arbitrary object in the game scene that defines the way the user sees the action on screen

cascading style sheets (CSS)—a computer language used to describe the presentation of structured documents that declares how a document written in a markup language such as HTML (Hypertext Markup Language) or XHTML (Extensible Markup Language) should look

casual gamer—a person who plays for the sheer satisfaction of the experience and is less intense about the games he or she plays (as opposed to a core gamer)

character design document—a written document that specifically targets one character, often the avatar of a game, and describes that character's appearance and moveset

coin-op game—a coin-operated game, often enclosed in a box and set in an arcade

collision detection—a method in game programming used to make sure that when objects come in contact with one another they behave with causal response as they would in the real world

collision polygon—an invisible border outlining an object in Torque Game Builder that the game engine registers as the boundary for collision responses

computer graphics—anything of a visual nature that artists create using the computer as a tool

control statement—a programming statement that involves looping or branching in the program code, offering random arbitration or depth possibilities

core gamer—a person who routinely plays lots of games and plays for the thrill of beating games (as opposed to a casual gamer)

core mechanics—the particular rules by which a player plays a specific video game

crunch time—the more intense period of game production as developers get closer to deadline time, resulting in overtime and working obscene hours to hurry and get a project finished on time

design document—a written document that tells the team all the details of the game, including which levels and characters will be in the game and how the player controls will work

domain name—a text alias for one or more IP addresses

Doppler effect—the principle that states that as a loud sound source comes closer to the listener, the higher the sound gains, and the farther away the sound source moves from the listener, the softer the sound gets until it fades away completely

draw order—a process used mainly to determine what objects are in front of others, which are behind others, and when one object collides with another

Easter egg—an industry name for a secret reward in a game, something few people but core gamers will find

emotioneering—a cluster of techniques seeking to evoke in gamers a breadth and depth of rich emotions to keep them engaged in playing a game

feng shui—the theory that people's moods and reactions can be orchestrated by the placement of objects in a room or (in the case of level design) on the computer screen

File Transfer Protocol (FTP)—a set of instructions that allows you to upload and download files to and from a web server

flowchart—a schematic representation of an algorithm or other step-by-step process, showing each step as a box or symbol and connecting them with arrows to show their progression

fog of war—a darkened screen feature of many strategy games, which is opened up as the player travels the terrain—therefore offering uncertainty as to the placement of enemy units and resources on that terrain before they are approached

Foley sounds—sounds that are not natural but are recorded custom to emphasize sounds that should be heard in context

function—the most common statement within an object-oriented programming language and a part of sequential logic

game concept—a short description of a game detailed enough to start discussing it as a potential project

game loop—a cycle of repetitive steps the player takes to win at any given game challenge

game pace (or flow)—the speed at which a player makes interactive actions and is guided through the game

game proposal—a written document intended to knock the socks off potential publishers and investors, that puts a game in its very best light

game script—a written document used for a complete overview and for one purpose not covered by other design documents: to target the key rules and core mechanics of gameplay; the theory behind playing the game should be present and enough details given that the reader could potentially play the game in their head

game testing—often done iteratively to ensure there are no glaring mistakes; this means that testing is done every step along the way and after a mistake is fixed the game is tested again to make sure the fix didn't break something else

game treatment document—a written document that sums up the core ideas behind a game but with more technical gameplay details than a high concept document would have

game—any activity conducted in a pretend reality that has a core component of play

gameplay—the way a game is played, especially in the way player interaction, meaningful direction, and an engaging narrative come together to entertain the player

gimmick—a clear and representable image of an idea; in level design, gimmicks are archetypal level types that are immediately memorable for players because of their familiar themes

gold master—the final edition of a game with all the bugs removed

graphical user interface (GUI)—the look of the shell extension of a game, including the windows, interactive menus, and heads-up display

green light—when you hear this, it means that a game development group has completed the preproduction phase, the required tools and finances to begin proper game creation have been acquired, and that the team is now geared up to start development in earnest

high concept document—a written document that highlights the key elements of a game in bite-size chunks for the purposes of grabbing producer or investor interest

high concept statement—a two-to-three sentence description of a game, akin to film/TV blurbs

Huizinga's Magic Circle—a concept stating that artificial effects appear to have importance and are bound by a set of made-up rules while inside their circle of use

human-machine interfacing (HMI)—the way in which a person, or user, interacts with a machine or special device, such as a computer

if-else statement—a programming statement that checks against a variable before executing a code

immersion—a key element of a game's popularity that creates addictive game play by submerging players in the entertainment form; with immersion, you get so engrossed in a game that you forget it's a game

Internet protocol (IP) address—a number that uniquely identifies any machine connected to the Internet, including computers, printers, and mobile devices

Internet—a global network of networks

laserdisc technology—an early forerunner to CD-ROM and DVD systems

level design—the creation of environments, scenes, scenarios, or missions in an electronic game world

levels—basic units, like chapters in a book, used for subdividing and organizing progress through a game

loop statement—a programming statement that makes all the code within the loop execute repeatedly, seemingly into perpetuity, until an exit condition is met

ludology—the academic study of games for the features that are distinctly related to play, including rules and simulation

milestone—a realistic step of production that can be accomplished on a game production timeline

motion cycle—a looping animation of a character or other object going frame-by-frame through its motions

motion emphasis—the process of slowing down and over-exaggerating the character action you want performed in order for audiences to keep up with and understand what is happening

moveset—a list of animations documenting a character's movements for a game, often found described in a character design document

non-player characters (NPCs)—characters not controlled by the player but by an artificial intelligence programmed into the game itself

object—in programming with TorqueScript, this can be any item in your game world

online communities—websites designed to foster communication and networking; they've been compared to bulletin boards, social clubs, and schoolyards

parallax scrolling—a method in 2D sidescrolling games whereby dissimilar planes of graphics scroll by at unlike rates of speed depending on their perceived relation to the viewer, to create the illusion of depth

particle effect—a group of similar graphics that combine to make an interesting visual effect; they are commonly used for things like explosions, rain, smoke, and several other cool animated special effects

pixel—a tiny square of color, one of many that make up a bitmap, that has a dot of red, green, and blue information in it that sets the color tone for that square

play—any grouping of recreational human activities, often centered around having fun

player interaction—a complex human-computer interface where the player gives her input or feedback to the game engine and the engine responds

proportionally; this interaction can reside on mouse and keyboard or on a handheld game controller, but it comes in the form of key or button combinations and directs the course of action in a game

portfolio—a list, often visual in nature, of what a designer has accomplished in their career thus far

postproduction phase (of game development)—during this development phase, testing, quality assurance, and bug-fixing are initiated, followed by a public relations campaign to get a game noticed by its target audience

predictive sound—a sound technique used in games that involves the placement of certain sounds before an event actually takes place

preproduction phase (of game development)—the phase that takes place before designers ever get their tools out and get started

production phase (of game development)—during this phase of development, the artists design the assets, characters, and environments on their computers, the writers set out dialogue and scripted events, the programmers code the controls and character behaviors, and the leaders make sure no one walks off the job

programming language—functional, imperative, and object-oriented (class-based) coding language used to develop software applications

protocols—common sets of rules that determine how computers communicate with each other over networks

prototype—a short playable demo of a game

quality assurance (QA)—apart from game testing and beta testing, testing must be done to look at the game as a whole and check it against the initial concept for consistency

quests—a special set of challenges that take place in both stories and games, thus linking narrative and play

ramping—a game gets increasingly harder the longer someone plays it

randomization—a method by which a computerized system can change the way in which a game is played

rapid iterative development—a production method where designers test ideas daily, see what's working and what's not, and abandon hurdles that are too difficult to get over

reactive environments—the game world responds to the player in logical and meaningful ways that help immerse the player in that game world

replayability—the "sweet spot" for game designers, where the player doesn't play the game only once through but wants to play the game repeatedly, either motivated by the need to excel or by the sheer excitement that comes from experiencing a compelling narrative

role—the part a player plays in a game, especially a role-playing game, often reflected as an avatar

role-playing games (RPGs)—games where the main goal is for the player to gain enough experience or treasure for completing missions and beating monsters to make her near-infinitely customizable character stronger

scratchware—a term coined at the time of the writing of the Scratchware Manifesto, meaning any game that is completely original, has great gameplay and replayability, runs well with few glitches, costs consumers less than $25 to purchase, and was created by less than three people

search engine—a program that searches the web for specific keywords and returns a list of documents in which those keywords are found

shooters—games in which the characters are equipped with firearms and focus on fast-paced movement, shooting targets, and blowing up nearly everything in sight

sidescrolling—a visual technique in games where the player's character starts on the left-hand side of the screen (usually) and the player navigates the character to the right-hand side of the screen; the invisible camera that the game is viewed from is locked onto the player character, following its movements

sound effects (SFX)—short recorded sounds that are interjected relative to visual effects to enhance the whole experience and give aural clues to what's going on onscreen

spawning—making an object appear in-game at a specific or random location, often referring to the dynamic generation of enemies or pick-up items

sprite—a 2D rectangular bitmap image that makes up most of the visual elements in a 2D game

static sprite—a single sprite image that consists of a non-animating character, object, or element

story/level progression document—a written document that records the storyline of a game and the way that the levels will progress from one to the next

strategy games—mental-challenge-based games, where the players build an empire, fortress, realm, world, or other construct, manage the resources therein, and prepare against inevitable problems like decay, hardship, economic depravity, revolution, or foreign invaders

switch statement—a programming statement that checks against a variable and then executes a code based upon context cases

syntax—applied linguistics

target market—a specific group of people to sell to

technical design document—a written document created by the lead programmer on a project, using the game script as a springboard to describe to the technical staff how a game should be built or implemented in the software

text-based game—in this type of game, players read descriptions of scenes and characters and select where they want to go and what they want to do from a list of choices, much like *Choose Your Own Adventure* or *Fighting Fantasy* books

tilemap—any regularly spaced grid that reuses the same set of tiles

tiles—flat 2D images, which can be static or animated, that are repeated over and over in the background to make up the terrain or surface area of the game world

triple-A (AAA) title—a game that sells big, in other words has a high cost to make and a higher return-on-investment; refers to an individual title's success or anticipated success if it is still in development

variable—any observable attribute in a programming language that changes its values when ordered to

voiceovers (VOs)—sounds done by artists recorded reading dialogue and narration scripts in a recording studio for purposes of providing spoken dialogue and narration in games

WAV files—generally uncompressed files, meaning that they can be quite large in size and sound pretty good; the quality of a WAV file is determined by how well it was originally recorded or converted

web browser—a software program that is used to locate and display web pages

web pages—specially formatted documents created using languages such as Hypertext Markup Language (HTML)

web server—a computer that is hooked up to the Internet 24/7 that might have one or more websites stored on it at any given time

world design document—a written document that describes the background information and the sorts of things that the make-believe world will contain in a general overview

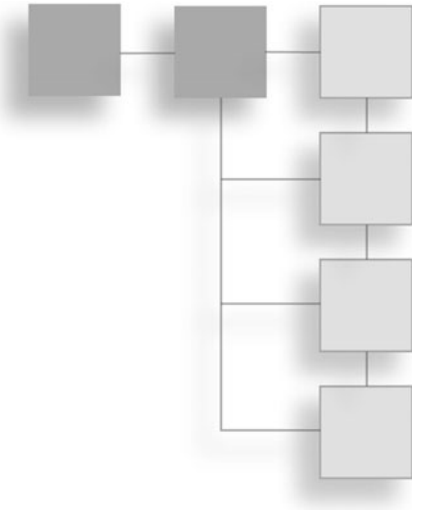
world limit—an invisible bounding box for in-game objects in Torque Game Builder

World Wide Web (WWW)—a subset of the Internet that supports web pages

X coordinate plane—a dimensional plane that goes lengthwise across the screen horizontally, much like longitude on a map

Y coordinate plane—a dimensional plane that goes heightwise across the screen vertically, much like latitude on a map

APPENDIX D



Online Resources

All of the following websites will be a boon to you as you start your own game design using Torque Game Builder.

Resources

110MB (http://110mb.com): a free web host service

50Webs (http://www.50webs.com): a free web host service

Acoustica (http://www.acoustica.com/sounds.htm): an online sound effects library

Adobe Dreamweaver (http://www.adobe.com/products/dreamweaver): the main info site for Adobe's web design software

Adobe Products (http://www.adobe.com/products): site to find and download the Adobe Illustrator program

Artist Launch (http://www.artistlaunch.com): an indie music provider

AtSpace (http://www.atspace.com): a free web host service

Audacity (http://audacity.sourceforge.net): the site where you can find info and download the free open source Audacity sound editor program

Blogger / Blogspot (http://www.blogger.com): a free blogging service

Brothersoft (http://search.brothersoft.com/tile-seamless): search results for seamless tile generators

Byethost (http://www.byethost.com): a free web host service

CD Baby (http://www.cdbaby.com): an indie music provider

CDBurnerXP (http://www.cdburnerxp.se): a CD burning software program

Cheetah Burner (http://www.cheetahburner.com): a CD burning software program

Dotster (http://www.dotster.com): a web domain name service

Files Upload (http://www.files-upload.com): a place online that will store your data files

FreeHostia (http://freehostia.com/free_hosting.html): a free web host service

Freeman Games (http://www.freemangames.com): David Freeman's information about emotioneering

Freesound Project (http://freesound.iua.upf.edu): an online sound effects library

FreeWebs (http://freewebs.com): a free web host service

GarageGames (http://www.garagegames.com): creators of the Torque Game Engine and Torque Game Builder

GeoCities (http://geocities.com): a free web host service

GoDaddy (http://www.godaddy.com): a web domain name service

Great Games Experiment (http://www.greatgamesexperiment.com): online gaming community

i3log.com (http://www.i3log.com): a free blogging service

Indiespace (http://www.indiespace.com): an indie music provider

Inkscape (http://www.inkscape.org): site to find and download the free open source art program Inkscape

Instant Action (http://www.instantaction.com): an indie game promotional tool and community

jEdit (http://www.jedit.org) with TIDE (http://torqueide.sourceforge.net): site of a syntax editor for use with TorqueScript

Jumpline (http://www.jumpline.com): a free web host service

Koumis Productions (http://www.koumis.com/soundfx.htm): an online sound effects library

Ljudo (http://www.ljudo.com/default.asp?lang=tEnglish&do=it): an online sound effects library

Media Tracks Free Sound Effects (http://www.media-tracks.com/default.asp?ID=5): an online sound effects library

Microsoft Expression (http://www.microsoft.com/expression): the main info site for Microsoft's web design software

Mojo Audio (http://www.mojoaudio.com): a provider of royalty-free sound effects for games

MP3.com (http://www.mp3.com): an indie music provider

Music4Games.net (http://www.music4games.net): an indie music provider

MyDataBus (http://www.mydatabus.com): a place online that will store your data files

MyDomain.com (http://www.mydomain.com): a web domain name service

MySpace (http://www.myspace.com): a giant online community

Nero (http://www.nero.com): a CD burning software program

Nvu (http://www.nvu.com): a free open source web design program

PBWiki (http://pbwiki.com): community wiki creation, useful for game design documentation

Register.com (http://www.register.com): a web domain name service

Roxio (http://www.roxio.com): a CD burning software program

Sickhead Games (http://www.sickhead.com): creators of Torsion, the TorqueScript syntax editor

Sound Effects Library (http://sound-effects-library.com): an online sound effects library

SoundClick (http://www.soundclick.com): an indie music provider

Stonewashed SFX (http://www.stonewashed.net/sfx.html): an online sound effects library

TorqueDev, a.k.a. CodeWeaver (http://tdn.garagegames.com/wiki/TorqueScript/IDE/Guide/TorqueDev): site of a syntax editor for use with TorqueScript

Tribal IDE (http://www.garagegames.com/index.php?sec=mg&mod=resource&page=view&qid=1413): site of a syntax editor for use with TorqueScript

Tripod (http://www.tripod.com): a free web host service

Wiki Upload (http://www.wikiupload.com): a place online that will store your data files

Windows Live Spaces (http://home.services.spaces.live.com): a free blogging service

Wordpress.com (http://www.wordpress.com): a free blogging service

XNA Creator's Club (http://msdn.microsoft.com/xna/creators): for use with Xbox 360 game development

Yahoo 360 (http://360.yahoo.com): a free blogging service

INDEX

A

A Prince of Our Disorder, **141**
AAA (triple-A) game title, 8
Acoustica sound library, 235
action games, 38–39
active tile, 59
addition (+) character, 274
addresses and domains, 257–258
Adobe Dreamweaver CS3 Revealed **(Bishop), 258**
Adobe Dreamweaver web site-construction kit, 258–259
Adobe Illustrator
- Ellipse tool, 85
- fills, 86
- Flare tool, 85
- guides in, 84
- Live Paint tool, 91–92
- Outline mode, 82, 84
- panel docks, 82
- panels, 82
- Pen tool, 88–90
- Polygon tool, 85
- Preview mode, 82, 84
- Rectangle tool, 85
- Rounded Rectangle tool, 85
- rulers in, 84
- Star tool, 85
- strokes, 86
- transform tools, 87–88
- trial version, 78
- uses for, 79–80
- Welcome Screen, 80–81
- workspace in, 81

Adobe website, 78
advertainment games, 42
advertisement
- online
  - addresses and domains, 257–258
  - basic description, 253
  - blogs, 265
  - design do's and don'ts, 261–262
  - domain names, 260
  - FTP, 262
  - getting your site notices, 262–264
  - Internet connections, 254–255
  - meta content, 264–265
  - MySpace accounts, 267–270
  - online communities, 265
  - prepping your images, 255–256
  - search engine optimization, 263–264
  - text content, prepping, 256
  - uploading your files, 262
  - URL addresses, 258
  - web browsers and web servers, 256–257
  - Web development, 255
  - websites, building and maintaining, 258–260
  - web-host services, 260–261
- proposals, 252–253
- self promotion, 249–250
- using gimmicks, 250–252

Age of Empires, **41**
Age of Time, **67**
Agent 47, 137
AIFF file format, 238
alchemical challenges, 175
Alcorn, Alan, 32

algorithms
control statements, 109–111
data types, 108
flowcharts, 111–112
functions, 108–109
human logic and, 106
objects, 108
operators, 108
variables, 107
AltaVista search engine, 263
amplitude, 232
anchor points, 85
And (&&) character, 274
animated characters
creating in TGB, 148–172
Disney principles, 145–146
motion cycle, 147–148
motion emphasis, 146
walk cycles, 147–148
animated sprites, 60–61, 95–97
animation datablocks, 60–61
***The Animator's Reference Book* (Pardew and Wolfley), 148**
***The Animator's Survival Kit* (Williams), 148**
AOL Search search engine, 263
appearance, character, 141
art bible, 19
art style, character design, 143
artificial life games, 42
artist
job description, 8
level, 9
user interface, 9
Artist Launch sound resource, 236
Ash, 140
Ask.com search engine, 263
***Atomize*, 54**
AtSpace web-host service, 260
Audacity audio editing program, 238–240
audience, defining the, 15
audio. *See* sound
autorun files, 248
avatar. *See* character design
Avellone, Chris, 12, 28

B

background elements, 77
background images
adding and scaling, 129–130
layering, 130–137
Bailey, Dona, 33
Barksdale, Karl (*Web Design BASICS*), 255
***Bejeweled*, 16, 42**
beta testing, 24
Bezier curves, 79–80
***BioShock*, 48**
Bishop, Sherry (*Adobe Dreamweaver CS3 Revealed*), 258
bit, 73
bitmaps, 30
Blair, Preston (*Cartoon Animation*), 148
Blizzard Entertainment, 4
***Blockland*, 67**
Blogger/Blogspot blog, 265
blogs, 265
***Bloodrayne*, 144**
BlueHost web-host service, 261
Bluth, Don, 34
boolean variables, 105
Brain Editor Professional syntax editor, 115
briefings, level design, 124
***Buccaneer: The Pursuit of Infamy*, 67**
burning application, 247–248
Burrows, Marcia, 126
Burton, Tim, 143
Bushnel, Nolan, 32
business, starting your own, 26–28
Byethost web-host service, 260
byte, 73

C

camera system (TGB), 59, 119–120
Campbell, Joseph, 48
***Cartoon Animation* (Blair), 148**
cartridge systems, 34
cascading style sheet (CSS), 260
case-sensitivity, TorqueScript programming language, 104
casual gamer, 15
casual games, 42
CD Baby sound resource, 236
CDBurnerXP application, 247
CD-burning applications, 247–248
Cell mode, 135
***Centipede*, 33–34**
challenges
alchemical, 175
clue-driven, 175
conflict, 178
fetch quest, 46

hint of reward element, 174
how to create, 46–49
lock mechanism, 175
maze, 175
mental deduction, 175
monster, 175
quests, 47–49, 176–177
rules-bound, 176
social/dialogue, 176
timing/sequence, 176
trap, 176
unfair, 49
character design. ***See also*** **enemies**
animated
creating in TGB, 148–172
Disney principles, 145–146
motion cycle, 147–148
motion emphasis, 146
walk cycles, 147–148
appearance, 141
art style consideration, 143
colorful, 143
exaggerated, 143
five lessons of design, 141–144
idiosyncratic, 138
impersonation, 143
as legend, 137
lucky accidents, capitalizing on, 142–143
name, 139–140
NPCs, 41
personality, 140–141
stereotypes, staying away from, 144
character design document, 21
character gimmicks, 251
chase/escort quest, 177
Cheetah Burner application, 247
Chess, 42
Choose Your Own Adventure**, 30**
classes, 106
closed paths, 85–86
Cloud, 137
clue-driven challenges, 175
CMYK (cyan, magenta, yellow, and black), 81
code
collision responses, 189–190
competition, 193–195
enemies
destroying player if touched, 208–209
enemy.cs file, 209–211
programming the, 201–202
spawning at random position, 206
in game.cs file, 151–152
movement commands, 153–156
new class, 152
player.cs file, linking to game.cs file, 157–158
reward
creating the, 182–185
programming the, 180–182
reward.cs file, 195–196
sound
descriptions, 240–241
dynamic, 241–242
profiles, 241
speed boosts, 166–167
speed variables, 159–160
updates, automatic, 161–165
weapons, 218–220
CodeOnce, 62
CodeWeaver syntax editor, 114
coin-op games, 31, 33
colleges, 270–272
collision
collision polygons, 185–188, 212–213
responses, 189–190
with TGB, 63–64
colorful character design, 143
combat quest, 177
Commercial License (TGB), 55
company, starting your own, 26–28
competition, 190–195
compressed audio, 237
computer graphics. ***See*** **graphics**
Computer Space**, 32**
concept. ***See*** **game concept**
conditional logic (?) character, 274
conflict-based challenge, 178
ConTEXT syntax editor, 115
control statements
if-else, 109–110
loop, 111
switch, 111
coordinates
sprite, 94
X, 74–75
Y, 74–75
core gamer, 15
core mechanics, 3–4
CorelDraw application, 78
corner anchor points, 85
Costikyan, Greg, 27
Crawford, Chris, 178
crawler, 263

creativity, game ideas, 13–14
Crimson Editor, 115
***Crimson Room*, 39**
Croft, Lara, 137, 144, 251
CSS (cascading style sheet), 260
curiosity, depths of immersion, 45
curly braces ({ }), 154
cyan, magenta, yellow, and black (CMYK), 81
***Cyclomite*, 69**

D

***Dark Horizons: Lore Invasion*, 67**
data types, 108
datablocks, 105
dB (decibel), 232
defense, 215–217, 248
depth information, 77
***Desert Gunner*, 67**
design. *See also* level design
  postproduction phase, 24–26
  preproduction phase, 19–22
  production phase, 23–24
design document
  character design, 21
  description of, 19
  game script, 21–22
  game treatment, 21
  high concept, 21
  importance of, 20
  story and level progression, 21
  technical design, 21
  world design, 21
design resolution, 119–120
designer
  artist, 8
  importance of, 7
  job classifications/descriptions, 8–9
  kinship between, 11
  leader, 9
  level artist, 9
  personality, 11
  portfolio, 10
  programmer, 9
  skills useful to, 10–11
  sound engineer, 9
  user interface artist, 9
  writer, 9
***Deus Ex: Invisible War*, 178**
***Diablo*, 4, 251**
dialogue/social challenges, 176
dial-up modem, 254
diamond node, 112
difficulty, level design, 124
Digital Equipment Corporation, 31
digital graphics. *See* graphics
digital sound, 237
Digital Subscriber Liner (DSL), 254
***Dimenxian*, 67**
***Diner Dash*, 42**
Dini, Dino, 4
directional handles, 85
Disney principles of animated characters, 145–146
division (/) character, 274
DNS (Domain Name System), 257–258
document. *See* design document
domains and addresses, 257–258, 260
Donkey, 140
***Donkey Kong*, 33–34**
***Donkey Kong Country*, 35, 123, 251**
***Doom*, 35**
Doppler Effect, 233
Dot5 Hosting web-host service, 261
Dotster domain name service, 260
***Dragon's Lair*, 34**
draw order, 77–78
DSL (Digital Subscriber Line), 254
***Dungeons and Dragons*, 4, 39, 140**
duration, level design, 124
dynamic sound, 241–242
dynamic sprites, 76
Dynamix company, 53

E

***Earthsiege*, 53**
Easter egg (secret reward), 15–16
echo command, 105
Ecko, Marc, 45
Eclipse syntax editor, 115
editing sound, 239
EditPlus syntax editor, 115
education, 270–272
edutainment games, 42
effects
  description of, 57
  sound, 234, 239
***Elder Scrolls IV: Oblivion*, 40**
Electronic Arts Game Innovations Lab, 14
electronic games
  advantages of, 3
  electronic game firsts, 30–34
  history of, 30

ellipse node, 112
Emacs syntax editor, 115
emitters, 57
emotioneering, 45–46
empathy, depths of immersion, 45
ending blocks, 105
ending statements, 105
enemies
basic creation, 201
collision polygon, 212–213
destroying player if touched, 208–209
as gimmick, 251
player fighting back, 215–217, 248
programming the, 201–202
spawning at random position, 205–207
speed range, 203–204
world limits, 204–205
engineer, sound, 9
Entertainment Software Association (ESA), 7
equal to (==) character, 274
ESA (Entertainment Software Association), 7
escort/chase quest, 177
***E.T. the Extra-Terrestrial*, 43**
***EverQuest*, 40**
exaggerated character design, 143
.exe file, 118
explosion, 225–228
exporting sound, 239–240
Expression application, 259

F

fairness, 49–50
Falstein, Noah, 11, 14
feasibility, 51
feedback, 50–51
feng shui, 128
Ferriter, Chris, 22
fetch quest, 46, 177
50Webs web-host service, 260
***Fighting Fantasy*, 30**
File Transfer Protocol (FTP), 262
fills, 86
***Final Fantasy* series, 138**
finite state machines (FSMs), 109
first-person shooters, 200
flipping, movement functions, 160
floating panels, 82
flowcharts, 112
***fog of war* terrain, 41**
Foley sound, 231
for statement, 111
***Fortune Tiles*, 54**
Foundation 9 company, 10
4 Fs of Great Game Design, 49–51
frames per second value, 149
FreeHostia web-host service, 260
Freeman, David, 45
Freesound Project sound library, 235
FreeWebs web-host service, 260
frequency, 232
FrontPage web-authoring program, 259
frustration, 48
FSMs (finite state machines), 109
FTP (File Transfer Protocol), 262
Fullerton, Tracy, 5, 14
fun (4 Fs of Great Game Design), 49
functions, 105, 108–109

G

gain, 232
game challenges. *See* challenges
game concept
defining the audience, 15
examples of, 17–19
high concept statement, 17
ideas, documenting, 15–16
what to include in, 17
game design. *See* design
game designer. *See* designer
game flow
gameplay and, 4
level design, 128
game loop, 47
game script, 21–22
game treatment document, 21
game.cs file, 157–158
gameplay
balanced design, 6
basic definition, 4
elements of, 5–6
graphics and, 6–7
sandbox style of, 30
gamer, 15
GarageGames company. *See also* TGB
founders of, 53
TGE (Torque Game Engine), 66–67
TGEA (Torque Game Engine Advanced), 69
Torque X Engine, 66–67

genre
action games, 38–39
adventure games, 39
advertainment games, 42
artificial life games, 42
casual games, 42
puzzle games, 42
RPGs (role-playing games), 39–41
serious games/edutainment games, 42
strategy games, 41–42
GeoCities web-host service, 260
***Getting Up: Contents Under Pressure*, 45**
GGE (Great Games Experiment), 26, 266
GIF (Graphics Interchange Format), 256
Gift, Tim, 53
gimmicks
advertisement, 250–252
character, 251
enemy, 251
fresh and new ideas, 126
graveyard level, 126
lava level, 125
level design, 125–128
location, 251
mine card ride, 126
overuse, 125
random element, 251
sewer level, 125
snow level, 126
underwater level, 126
weapon, 251
Glassner, Andrew, 47
global variables, 107, 274
goals, level design, 123–124
GoDaddy domain name service, 260
***Gold Fever*, 54**
gold master, 25
***Golden Fairway*, 67**
Google search engine, 263
***Grand Theft Auto: San Andreas*, 124**
***Grand Theft Auto* series, 4–5, 30, 141**
graphics. *See also* Adobe Illustrator
2D *versus* 3D game, 35–37
bit, 73
byte, 73
coordinates, 74–75
digital, 71–72
draw order, 77–78
gameplay and, 6–7
JPEG format, 75
pixel, 73–74
PNG format, 75
sprites, 30, 76
SVG, 80
in TGB, 93–95
tiled images and backgrounds, 77
vector art, 78–79
Graphics Interchange Format (GIF), 256
graveyard level gimmicks, 126
Great Games Experiment (GGE), 26, 266
***Grim Fandango*, 143**
GUI Editor (TGB), 58–59
guides, Adobe Illustrator, 84
Gygax, Gary, 39

H

***Half-Real: Video Games between Real Rules and Fictional Worlds*, 48**
***Halo 3*, 7–8**
height, sprite, 95
hero's journey, 48
Hertz (Hz), 232
high concept document, 21
high concept statement, 17
hit points, 40
HMI (human-machine interfacing), 102–104
Holmes, Marc Taro, 123
Hosting Department web-host service, 261
HostMonster web-host service, 261
HostPapa web-host service, 261
Hotbot search engine, 263
Howard, Jeff (*Quests: Design, Theory, and History in Games and Narratives*), 176–177
HTML (Hypertext Markup Language), 254
Huizinga, Johan (*Huizinga's Magic Circle* concept), 3
human logic, 106
human-machine interfacing (HMI), 102–104
Hypertext Markup Language (HTML), 254

I

i3log.com blog, 265
id Software, 35
ideas
brainstorming, 12–14
coming up with, 12–15
creativity in, 13–14
documenting, 15–17
game concept, 15–19

identification, depths of immersion, 45
if-else statement, 109–110
illusion of depth, 61
***The Illusion of Life: Disney Animation* (Johnston and Thomas), 145**
Illustrator. *See* Adobe Illustrator
image files
loading into TGB, 94, 96–97
resizing, 150
immersion, 45
impersonation, character design, 143
independent game movement, 26–28
Indie License (TGB), 55
Indiespace sound resource, 236
inheritance, 105
InMotion Hosting web-host service, 261
The Inspiracy, 11
installing TGB, 118
InstantAction, 69
intensity, sound, 232
interactivity
cinematic scene, 43
emotioneering, 45–46
empowering the player, 44
immersion, 44–45
reactive environment, 44
suspension of disbelief, 45
Internet advertisement. *See* online advertisement
IP addresses, 257–258
isometric perspective, 34, 41
ISP (Internet service provider), 254

J

jEdit syntax editor, 114
Jen's File Editor, 115
job descriptions, 9
Johnston, Ollie (*The Illusion of Life: Disney Animation*), 145
JPEG (Joint Photographic Experts Group) format, 75, 256
Jumpline web-host service, 261

K

***Kachinko*, 54**
keyboard events, 275
keyboard shortcuts, 273–274
keywords, 264
***King Kong: Skull Island Adventure*, 54**
***King's Quest* series, 47**
Kirby, 137
Koumis Productions sound library, 235

L

laserdisc technology, 34
lava level gimmicks, 125
layers
background images, 130–137
description of, 61
leader job description, 9
legends, 137
***LEGO: Bricktopia*, 54**
leitmotiv, 233–234
Lemmy syntax editor, 115
less than (<) character, 274
less than or equal to (<=) character, 274
level artist job description, 9
Level Builder (TGB), 57
level design. *See also* design
background images
adding and scaling, 129–130
layering, 132–137
basic description of, 123
briefings, 124
difficulty, 124
do's and don'ts, 128–129
duration, 124
game flow, 128
game world consideration, 128
gimmicks, 125–128
goals, 123–124
level structure, 123
saving levels, 122
scale, 124
Levy, Hope, 235
libraries, sound, 235–236
license
licensed games, 42–43
TGB, 55–56
licensed property (LP), 43
links and link-backs, 264–265
LivePaint tool, 87, 91–92
Ljudo sound library, 235
local variables, 107, 274
location gimmicks, 251
lock mechanism challenges, 175
Logg, Ed, 33
loop statement, 111
***The Lord of the Rings*, 48**
LP (licensed property), 43

ludology, 3
Lycos search engine, 264

M

Mack, John E., 141
***Magecraft*, 67**
***Marble Blast*, 26**
***Marble Blast Gold*, 67**
***Marble Blast Online*, 69**
Mario, 137, 251
***Mario 64*, 140**
Massachusetts Institute of Technology (MIT), 31
***The Matrix!*, 3**
maze challenges, 175
Media Tracks Free Sound Effects sound library, 235
Meier, Sid, 4
mental deduction challenges, 175
meta content, 264–265
metasearch engines, 263
Mette, Justin, 26
Mikami, Shinji, 36
mine cart ride gimmicks, 126
***Minigold Mania*, 67**
***Minions of Mirth*, 68**
***Mission Stencil Story*, 23**
MIT (Massachusetts Institute of Technology), 31
Mitchell, Billy, 33
Miyamoto, Shigeru, 33, 142
Mojo Audio website, 235
monster challenges, 175
Moore, Michael E., 5
***Mortal Kombat*, 6**
motion cycle, 147–148
motion emphasis, 146
mount forces, 65
movement commands, 153–156
  automatic updates, 160–165
  flipping, 160
  speed boosts, 166–167
  speed variables, 159–160
  world limit settings, 166–167
MP3 file, 237
MP3.com sound resource, 236
MSN Live Search search engine, 264
MultiEdit syntax editor, 115
multiplication (*) character, 274
music. *See* sound
Music4Games.net sound resource, 236
MyDomain.com domain name service, 260
MySpace accounts, 267–270

N

name
  character, 139–140
  project, 121
negative/positive reinforcement, 50
Nero burning application, 247
***Neverwinter Nights*, 39**
***Nintendogs*, 42**
non-player characters (NPCs), 41
not (!) character, 274
not equal to (!=) character, 274
NPCs (non-player characters), 41
NURBS, 79
Nvu web-authoring application, 259–260

O

objects, 105, 108
Obsidian Entertainment, 12
O'Connor, Greg, 229
ODP (Open Directory Project) search engine, 264
OGG files, 237–238
***Once Upon A Time*, 68**
110MB web-host service, 260
online advertisement
  addresses and domains, 257–258
  basic description, 253
  blogs, 265
  design do's and don'ts, 261–262
  domain names, 260
  FTP, 262
  Internet connections, 254–255
  meta content, 264–265
  MySpace accounts, 267–270
  online communities, 265
  prepping your images, 255–256
  search engine optimization, 263–264
  text content, prepping, 256
  uploading your files, 262
  URL addresses, 258
  web browsers and web servers, 256–257
  Web development, 255
  web-host services, 260–261
  websites, building and maintaining, 258–260
online communities, 265
online shooters, 200

open command, 248–249
Open Directory Project (ODP) search engine, 264
open paths, 85
operators
basic description of, 108
lists of, 274–275
Or (||) character, 274
***Orbz*, 26, 68**
Outline mode (Adobe Illustrator), 82, 84
Overman, Rick, 53

P

pace. *See* game flow
packages, 105
Packaging Utility (TGB), 66, 246–247
***Pac-Man*, 33, 109–110**
panels, Adobe Illustrator, 82
parallox scrolling, 61
Pardew, Les (*The Animator's Reference Book*), 148
Particle Builder (TGB), 57
particle effect, 225–228
particles, 57
paths
closed, 85–86
open, 85
***PCD Music Lounge*, 68**
PDP-1 (Programmed Data Processor-1), 31
Pen tool (Adobe Illustrator), 88–90
***Penny Arcade Adventures: On the Rain-Slick Precipice of Darkness*, 68**
personality
character, 140–141
designer, 11
***Phantasia*, 54**
***PHP for Teens* (Sethi), 255**
physics engine (TGB), 65
***Pirate Tales*, 12–13**
***Pirates of the Caribbean*, 12**
pitch, 232
pixel-based art, 78
pixels, 30, 73–74
Pizer, Patricia, 25
***Planescape: Torment*, 139**
planetary state of being, 45
player
empowering the, 44
interaction, core mechanics, 3–4
player.cs file, 157–158
PNG (Portable Network Graphics) format, 75, 256
***Pokemon*, 140**
Poker Texas Hold 'em, 42
***Pong*, 32, 212**
Portable Network Graphics (PNG) format, 75, 256
***Portal*, 16**
portfolio, 10
positive/negative reinforcement, 50
postproduction phase
gold master, 25
QA, 25
testing and, 24–25
predictive sounds, 233
preproduction phase
art bible, 19
design document, 19–22
game proposal, 19
prototype, 19, 22–23
pretend reality, 2–3
Preview mode (Adobe Illustrator), 82, 84
***Prince of Persia: The Sands of Time*, 251**
***Principles of Web Design, 4th Edition* (Sklar), 255**
Pro License (TGB), 55
production phase
green light terminology, 23
postproduction, 24–26
preproduction phase, 19–22
process of, 23–24
rapid iterative development, 24
profiles, sound, 241
Programmed Data Processor-1 (PDP-1), 31
programmer job description, 9
programming code. *See* code
programming languages, 104
projects
importing resources into, 122–123
name, 121
new project creation, 120–122
saving new level, 122
promotion. *See* advertisement
proposal
description of, 19
pitching to publisher, 252–253
protocols, 254
prototype, 19, 22–23
***Psychonauts*, 126, 143**
publishers, pitching proposals to, 252–253
puzzle games, 42
***Puzzle Poker*, 54**

Q

QA (quality assurance), 25
quests
as challenges, 47–49
combat challenge, 177
escort/chase, 177
fetch challenge, 177
***Quests: Design, Theory, and History in Games and Narratives* (Howard), 176–177**

R

***Rack 'em Up Road Trip*, 54**
***Radarscope*, 33**
ramping, 124
random element gimmicks, 251
randomization, 4
rapid iterative development, 24
reactive environment, 44
recording sound, 239
Rectangle tool (Adobe Illustrator), 85
Register.com domain name service, 260
replayability, 4
***Resident Evil*, 36, 235**
resolution
design, 119–120
resolution-independent, 82
reward
creating the, 182–185
Easter egg (secret reward), 15–16
hint of reward element, 174
programming the, 180–182
reward.cs file, 195–196
***RocketBowl*, 68**
role-playing games (RPGs), 39–41
***Rolling Stone* magazine, 141**
Rosen, David, 31
rotation, sprite, 95
Rounded Rectangle tool (Adobe Illustrator), 85
Roxio burning application, 247–248
RPGs (role-playing games), 39–41
rulers, Adobe Illustrator, 84
rules-bound challenges, 176
Russell, Steve, 32

S

***Sachi's Quest*, 68**
sandbox style of gameplay, 30
saving new levels, 122
Scalable Vector Graphics (SVG), 80
scale, level design, 124
scene graph (TGB), 62
Schafer, Tim, 126, 143
schools, 270–272
***Scratchware*, 27**
scripts. *See* code
SDK (software development kit), 57
search engine optimization, 263–264
secret reward, 15–16
SEGA logo, 31
Selection tool, 168–169
self promotion, 249–250
Sellers, Mike, 12
sequence/timing challenges, 176
sequential logic, 108–109
serious games/edutainment games, 42
Seropian, Alex, 69
Sethi, Maneesh
PHP for Teens, 255
Web Design for Teens, 255
sewer level gimmicks, 125
***Shadow Man*, 126**
***Shelled!*, 68**
shellexecute command, 249
***Shokrok Throwdown*, 68**
shooters
characteristics of, 199–200
first-person, 200
gameplay emphasis, 199
online, 200
shortcuts, keyboard, 273–274
***Shrek*, 140**
sidescrolling games, 56
***Sims*, 30**
size, sprite, 95
Sklar, Joel (*Principles of Web Design, 4th Edition*), 255
SlickEdit syntax editor, 115
smooth anchor points, 85
snow level gimmicks, 126
social/dialogue challenges, 176
software development kit (SDK), 57
Solitaire, 42
sound
AIFF file format, 238
amplitude, 232
Audacity audio editing program, 2
38–240
cinema, 229–231
compression, 237
descriptions, 240–241
digital, 237
dynamic, 241–242

editing, 239
effects, 234, 239
exporting, 239–240
Foley, 231
frequency, 232
gain, 232
influencing factors on, 232–234
intensity, 232
laws of, 232
leitmotiv, 233–234
libraries, 235–236
mixing your own, 236–237
MP3 files, 237
music, 234
OGG files, 237–238
pitch, 232
predictive, 233
profiles, 241
recording, 239
situational impacts on, 234
space's impact on, 232–233
static, 242
technological advances, 230–231
in TGB, 240–243
timbre, 232
time's impact on, 233–234
uncompressed audio, 237
voiceovers, 235
WAV file format, 237
Sound Effects Library, 236
sound engineer job description, 9
sound waves, 232
SoundClick sound resource, 236
SourceEdit syntax editor, 115
***Space Ace*, 34**
***Space Invaders*, 32–33**
***Spacewar!*, 32**
spawn() function, 182, 206
spawning enemies at random position, 205–207
Spector, Warren, 178–179
speed boosts, 166–167, 203–204
speed variables, 159–160
spell checking, 159, 197
spider, 263
Spock, 140
sprites
animated, 60–61, 95–97
coordinates, 94
defined, 30, 76
dynamic, 76
placing into game, 149
rotating, 95
static, 61, 76, 93–95
spyware, 261
square node, 112
***Star Trek*, 140**
***Star Wars*, 48**
***Starsiege*, 53**
startGame() function, 152
StartLogic web-host service, 261
statements
control, 109–111
ending, 105
for, 111
if-else, 109–110
loop, 111
switch, 111
while, 111
static sound, 242
static sprites, 61, 76, 93–94
stereotypical characters, staying away from, 144
Stonewashed SFX sound library, 236
story and level progression document, 21
strategy games, 41–42
strings
description of, 105
tagged, 108
strokes, 86
Stubbs, Todd (*Web Design BASICS*), 255
***Super Mario Bros*, 30, 33, 56, 251**
suspension of disbelief, 45
SVG (Scalable Vector Graphics), 80
switch statement, 111
sympathy, depths of immersion, 45
syntax editors, 113–115

T

tagged strings, 108
Takagi, Toshimitsu, 39
target market, audience as, 15
Tech Model Railroad Club (TMRC), 31
technical design document, 22
***Tennis Critters*, 26**
Terrano, Mark, 10
testing
beta, 24
postproduction phase, 24–25
spell checking, 159, 197
***Tetris*, 42**
text-based game, 30

TGB (Torque Game Builder)
2D Toolset
GUI Editor, 58–59
Level Builder, 58
Particle Builder, 57
Tile Editor, 59
animated sprites, 95–97
camera system, 59–60, 119–120
collision detection, 63–64
demo download, 117–118
discussed, 1
editing tilemaps in, 98–99
games made with, 54
installation, 118
layers, 61
licenses, 55–56
loading image files in, 94, 96–97
loading tilemaps into, 98
new project creation in, 120–122
Packaging Utility, 66, 246–247
parallax scrolling, 61
physics engine, 64–65
resources, 69
scene graph, 62
sidescrolling games, 56
sound in the, 240–243
sprite support, 60–61
Torque Community, 66
TorqueNet Lite feature, 65–66
TorqueScript scripting language, 62–63
using graphics in, 93–95
workspace, 118–120
TGE (Torque Game Engine)
description of, 53
games made with, 67–68
TGEA (Torque Game Engine Advanced), 69
***Thief: Deadly Shadows*, 178**
***ThinkTanks*, 68–69**
Thomas, Frank (*The Illusion of Life: Disney Animation*), 145
3D graphics
2D game graphics *versus*, 35–37
2D/3D hybridization, 36–37
in modern games, 37
Tile Editor (TGB), 59
tiled images, 77
tilemaps, 77
editing in TGB, 98–99
layout, 97
loading into TGB, 98
tiles, 59
timbre, 232
timing/sequence challenges, 176
TMRC (Tech Model Railroad Club), 31
Toast application, 248
***Tomb Raider*, 144, 251**
Torsion editor, 112–114
Torque Boot Camps, 55
Torque Game Builder. *See* TGB
Torque Game Engine. *See* TGE
Torque Game Engine Advanced (TGEA), 69
Torque X Engine, 66–67
TorqueDev syntax editor, 114
TorqueLite feature, 65–66
TorqueScript scripting language
algorithms, 106–110
boolean variables, 105
case-sensitivity, 104
classes, 105
datablocks, 105
description of, 62–63, 101
echo command, 105
ending blocks, 105
ending statements, 105
functions, 105
HMI, 102–104
inheritance, 105
objects, 105
packages, 105
strings, 105
variables, 105
transform tools (Adobe Illustrator), 87–88
transportation, depths of immersion, 45
trap challenges, 176
Tribal IDE syntax editor, 115
***Tribes*, 53**
***Tribes 2*, 53**
***Trick Ball*, 54**
triple-A (AAA) game title, 8
Tripod web-host service, 261
trivia games, 42
***TubeTwist*, 68**
Tunnel, Jeff, 53
Twintale Entertainment, 12–13

U

***Ultimate Duck Hunting*, 68**
UltraEdit syntax editor, 115
underscore (_), 107
underwater level gimmicks, 126
updates, setting automatic, 160–165
user interface artist job description, 9

V

variables
description of, 105
global, 107, 274
local, 107, 274
vector art, 78–79
vector-based art, 78
Vim syntax editor, 115
Virtual Fighter, **37**
viruses, 261
voiceover (VO), 235
Voodoo Vince, **126**

W

Wainess, Richard, 125
walk cycles, 147–148
WAV file format, 237
weapons, 216–219, 251
web browsers and servers, 256–257
Web Design BASICS **(Barksdale and Stubbs), 255**
Web Design For Teens **(Sethi), 255**
Web development, 255
Website, getting your site noticed, 262–264
web-host services, 260–261
Welcome Screen (Adobe Illustrator), 80–81
while statement, 111
width, sprite, 95
Wildlife Tycoon: Venture Africa, **68**
Williams, Josh, 69
Williams, Richard (*The Animator's Survival Kit***), 148**
Windows Live Spaces blog, 265
wireless capability, 254
Wixon, Dennis, 47
Wolfley, Ross (*The Animator's Reference Book***), 148**
Wolfstein 3D, **35**
Wordpress.com blog, 265
workspace, TGB, 118–120
world design document, 21
world limit settings, 166–167, 204–205
World of WarCraft, **25, 39, 143**
World Wide Web (WWW), 254
writer job description, 9
WWW (World Wide Web), 254

X

X coordinate, 74–75
XNA Creator's Club, 67
XNA Game Studio Express, 66–67

Y

Y coordinate, 74–75
Yahoo 360 blog, 265
Yahoo! search engine, 264
Yarbo, Hank, 48

Z

Z buffer, 77
Zaxxon, **34**
Zelda, 251
Zork, **39–40**

License Agreement/Notice of Limited Warranty

By opening the sealed disc container in this book, you agree to the following terms and conditions. If, upon reading the following license agreement and notice of limited warranty, you cannot agree to the terms and conditions set forth, return the unused book with unopened disc to the place where you purchased it for a refund.

License

The enclosed software is copyrighted by the copyright holder(s) indicated on the software disc. You are licensed to copy the software onto a single computer for use by a single user and to a backup disc. You may not reproduce, make copies, or distribute copies or rent or lease the software in whole or in part, except with written permission of the copyright holder(s). You may transfer the enclosed disc only together with this license, and only if you destroy all other copies of the software and the transferee agrees to the terms of the license. You may not decompile, reverse assemble, or reverse engineer the software.

Notice of Limited Warranty

The enclosed disc is warranted by Course Technology to be free of physical defects in materials and workmanship for a period of sixty (60) days from end user's purchase of the book/disc combination. During the sixty-day term of the limited warranty, Course Technology will provide a replacement disc upon the return of a defective disc.

Limited Liability

THE SOLE REMEDY FOR BREACH OF THIS LIMITED WARRANTY SHALL CONSIST ENTIRELY OF REPLACEMENT OF THE DEFECTIVE DISC. IN NO EVENT SHALL COURSE TECHNOLOGY OR THE AUTHOR BE LIABLE FOR ANY OTHER DAMAGES, INCLUDING LOSS OR CORRUPTION OF DATA, CHANGES IN THE FUNCTIONAL CHARACTERISTICS OF THE HARDWARE OR OPERATING SYSTEM, DELETERIOUS INTERACTION WITH OTHER SOFTWARE, OR ANY OTHER SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES THAT MAY ARISE, EVEN IF COURSE TECHNOLOGY AND/OR THE AUTHOR HAS PREVIOUSLY BEEN NOTIFIED THAT THE POSSIBILITY OF SUCH DAMAGES EXISTS.

Disclaimer of Warranties

COURSE TECHNOLOGY AND THE AUTHOR SPECIFICALLY DISCLAIM ANY AND ALL OTHER WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING WARRANTIES OF MERCHANTABILITY, SUITABILITY TO A PARTICULAR TASK OR PURPOSE, OR FREEDOM FROM ERRORS. SOME STATES DO NOT ALLOW FOR EXCLUSION OF IMPLIED WARRANTIES OR LIMITATION OF INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THESE LIMITATIONS MIGHT NOT APPLY TO YOU.

Other

This Agreement is governed by the laws of the State of Massachusetts without regard to choice of law principles. The United Convention of Contracts for the International Sale of Goods is specifically disclaimed. This Agreement constitutes the entire agreement between you and Course Technology regarding use of the software.